USING INTERIOR POINT ALGORITHMS FOR THE SOLUTION OF LINEAR PROGRAMS WITH SPECIAL STRUCTURAL FEATURES

KOSMAS GOSSIS

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

September 1995

THE UNIVERSITY OF ASTON IN BIRMINGHAM

USING INTERIOR POINT ALGORITHMS FOR THE SOLUTION OF LINEAR PROGRAMS WITH SPECIAL STRUCTURAL FEATURES

KOSMAS GOSSIS

Doctor of Philosophy

September 1995

Summary

Linear Programming (LP) is a powerful decision making tool extensively used in various economic and engineering activities. In the early stages the success of LP was mainly due to the efficiency of the simplex method. After the appearance of Karmarkar's paper, the focus of most research was shifted to the field of interior point methods.

The present work is concerned with investigating and efficiently implementing the latest techniques in this field taking sparsity into account. The performance of these implementations on different classes of LP problems is reported here.

The preconditioned conjugate gradient method is one of the most powerful tools for the solution of the least square problem, present in every iteration of all interior point methods. The effect of using different preconditioners on a range of problems with various condition numbers is presented.

Decomposition algorithms has been one of the main fields of research in linear programming over the last few years. After reviewing the latest decomposition techniques, three promising methods were chosen and implemented. Sparsity is again a consideration and suggestions have been included to allow improvements when solving problems with these methods. Finally, experimental results on randomly generated data are reported and compared with an interior point method.

The efficient implementation of the decomposition methods considered in this study requires the solution of quadratic subproblems. A review of recent work on algorithms for convex quadratic was performed. The most promising algorithms are discussed and implemented taking sparsity into account. The related performance of these algorithms on randomly generated separable and non-separable problems is also reported.

Key Words: Decomposition, Interior Point Methods, Least Squares, Linear Programming, Quadratic Programming

I wish to express my thanks to

Mr. George R. Lindfield, my supervisor, for his expert guidance and friendship whenever needed,

all members of staff of the Department of Computer Science & Applied Mathematics, and Information Systems, especially Neil Toyer and Mary Fenton, for their technical help and advice,

all my fellow research students, especially Amjit Bhatti for his stimulating ideas and lively debates,

Finally, I would like to thank my parents and all my friends for their understanding and support.

To my parents

Acknowledgements

I wish to express my thanks to the following people:

Mr. George R. Lindfield, my supervisor, for his support, guidance, and availability whenever needed,

all members of staff of the Department of Computer Science & Applied Mathematics, and Information Systems, especially Neil Toyer and Mary Finean, for their technical help and advice,

all my fellow research students, especially Amrit Hothi for his stimulating ideas and lively debates.

Finally, I would like to thank my parents and all my friends for their understanding and support

List of Contents

List of Figures

Figure **Page**

List of Tables

Chapter 1

Introduction

1.1 A Brief History

The requirement for methods of linear optimization arises from the necessity to analyse mathematical models describing the theory of systems, processes, equipment, and devices which occur in practice.

During the last 30 years the techniques of linear optimization have emerged as an important subject for study and research. The increasingly widespread application of optimization has been stimulated by the availability of digital computers and the necessity of using them in the investigation of large systems.

Linear Programming (LP) was established in 1947 with the design of the simplex method of G.B.Dantzig for solving optimum planning problems. A period of rapid developments and exciting discoveries in this new field followed and continue today. As noted in Salhi, (1987):

> "In the post-war era, LP has provided a good framework for the analysis of classical, economic theories such as the Walras Mathematical Model of Economy and Leontief Input-Output Model. It has also been successfully used to bring together different fields of pure mathematics, such as convex set theory, combinatorics, and two-person game theory."

From the beginning, the simplex method was effective on almost any type of LP problem and over the years the algorithm has been further polished and new variants of it

have been developed. In 1972 the simplex method was shown to run in exponential time for an artificial class of LP problems, [Klee & Minty, (1972)].

In 1979 the first polynomial time algorithm was announced, [Khachyan, (1979)]. In spite of the better complexity of this method (complexity will be discussed in the following section), the algorithm did not prove itself as well as the simplex method in practice. Computationally, in broad terms, there were two major difficulties with the method. The number of iterations tends to be very large, and the computation associated with each iteration is much more costly than a simplex iteration, [Dodani & Babu, (1990)].

In 1984 interest in linear programming was intensified by the publication of an interior point method that was not only polynomial in complexity but was also claimed by its inventor to be faster than the simplex method, [Karmarkar, (1984a); (1984b)]. The existence of problems for which the simplex method runs in exponential time and the appearance of polynomial time algorithms encouraged the debate over the efficiency of the simplex and a crucial question arose: Is LP in the P-class or NP-class?

Before going any further, it is important to define some terminology borrowed from complexity theory; the definition may help to see how this theory contributes to understanding algorithms and evaluating their performance.

1.2 Algorithm and Problem Complexity

Usually, for a given problem, a range of algorithms may be used to solve it. As a random choice may not be suitable, it is useful to have some criteria for identifying a specific algorithm. These criteria are the amount of CPU time and the storage required to run a code of the algorithm on a computer, [Lovász, (1984)].

One of the main concerns of complexity theory is to find, for a given algorithm, a bound on its running time, i.e., its time complexity function, and a bound on the space requirement, i.e., its space complexity function. Time is usually the only factor considered. However, the theory can be extended to include storage. In finding these bounds, the

problem's difficulty can also be investigated. This allows us to separate problems into different complexity classes. Hence, algorithm complexity and problem complexity are interrelated, although a distinction between them should be made. Algorithm complexity is the cost of a particular algorithm while problem complexity is the minimal cost over all possible algorithms, [Traub & Wozniakowski, (1982)].

Two complexity classes; the P-class and the NP-class, were already mentioned . The P-class, probably the most studied, contains problems for which a polynomial time algorithm has been found on deterministic computers (like the ones used in the real world). A polynomial time algorithm is one with running time bound, (worst case complexity), which is a polynomial function of the length of the problem data (e.g.., 2n, n^3+n, etc.) or behaves like one (e.g.., logn, nlogn, n^6logn, etc.), [Garey & Johnson, (1979); Kronsjö, (1985)]. For example, the algorithm of Gaussian eliminations for the solution of a system of linear equations is an $O(n^3)$ algorithm.

The NP-class contains problems for which a polynomial time algorithm can be found only on a non-deterministic computer. Non-deterministic computers are pure mathematical inventions. On real life computers only exponential time algorithms can be found for them. These algorithms have time bounds which are exponential functions or behave similarly, (e.g.., e^n+n, 2^n, etc.). An example of a problem in this class is the travelling-salesman problem whose time bound is O(n!). The hardest problems in the NP-class form the NP-Complete class. Intuitively, problems in the NP-class are of the form 'determine whether a solution exists'. Their complementary problems are of the form 'establish that there are no solutions' and constitute the CO-NP-class, [Kronsjö, (1985)].

As early as 1953, Von Neumann made the distinction between polynomial and exponential time algorithms. However, it was not until 1965 that the class of problems solvable by polynomial algorithms was identified, [Cook, (1983)]. This was due to Edmonds, (1965), who was the first to express the thought that exponential time computability approximately indicates how difficult a problem is. Consequently, he introduced the notion of "easy" and "hard" problems as well as "good" and "bad" algorithms.

In practical terms, this idea of classifying problems and algorithms is not totally justified. Indeed, many reliable and practical algorithms, such as the simplex method, are known to run in exponential time for some special cases, and many good algorithms in theory, such as the ellipsoid method, are inefficient in practice. It is in this respect that the average run time (average complexity) is relevant to understanding the behaviour of algorithms. However, average time bounds are more difficult to derive, as *a priori* probability distributions on the data must be postulated, [Lovász, (1984)].

Until the 1980s, the LP problem was believed to be in the NP-complete group. It was thought that the discovery of a polynomial time algorithm for LP would bring an answer to the outstanding question of whether P=NP. As will be seen in the following sections, such an algorithm has been discovered which shows that LP is in the P-class. However, a closer study of the problem's properties revealed that linear programming has the properties of the NP, as well as the CO-NP groups. Because there is strong evidence that NP≠CO-NP, LP can only belong in one of these groups. Further studies supported the argument that LP is not a member of the NP-class, [Garey & Johnson, (1979); Kronsjö, (1985)].

1.3 LP Problem Statement and Notation

The general linear programming problem and equivalent forms will be stated before going into details of the present work. The notation will be consistently followed in subsequent chapters. Other forms and symbols will be defined when introduced.

The General Form

The general problem of linear programming is the search for the optimum (maximum or minimum) of a linear function of variables subject to linear relations (equalities or inequalities) called constraints. Some constraints are specific to some or all variables, for

example, the non-negativity constraints ($x_j \geq 0$). Some or all variables can be arbitrary. It is, however, very common to impose *a priori*, the condition of non-negativity, on all variables in all economic problems.

According to the above definition, the algebraic formulation of the general LP problem, [Simonnard, (1966)], is as follows:

$$\text{min (or max)} \quad z = \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad i = 1, \ldots, p,$$

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \qquad i = p+1, \ldots, m, ,$$

$$x_j \geq 0 \quad j = 1, \ldots, p,$$

$$x_j \text{ arbitrary} \quad j = q+1, \ldots, n,$$

where a_{ij}, b_i, c_j, x_j and $z \in R$, for $i = 1, \ldots, m$, and $j = 1, \ldots, n$.

Henceforth this will be referred to as **GLP**.

Equivalent Formulations

The general LP problem can be put under more compact and easy to handle forms. These forms are equivalent.

The Canonical Form :

$$\begin{array}{ll} \text{Min } \mathbf{c^T x} & \mathbf{c} \in R^n \\ \text{s.t. } \mathbf{Ax} \geq \mathbf{b} & \mathbf{A} \in R^{m \times n}, \mathbf{b} \in R^n \\ \mathbf{x} \geq \mathbf{0} & \mathbf{x} \in R^n \end{array}$$

The Standard Form :

$$\begin{array}{ll} \text{Min } \mathbf{c^T x} & \mathbf{c} \in R^n \\ \text{s.t. } \mathbf{Ax} = \mathbf{b} & \mathbf{A} \in R^{m \times n}, \mathbf{b} \in R^n \\ \mathbf{x} \geq \mathbf{0} & \mathbf{x} \in R^n \end{array}$$

The Mixed Form :

$$\begin{array}{ll} \text{Min } \mathbf{c}^T\mathbf{x} & \mathbf{c} \in R^n \\ \text{s.t. } \mathbf{A}_1\mathbf{x} \geq \mathbf{b}_1 & \mathbf{A}_1 \in R^{m_1 \times n},\ \mathbf{b}_1 \in R^{m_1} \\ \quad \mathbf{A}_2\mathbf{x} = \mathbf{b}_2 & \mathbf{A}_2 \in R^{m_2 \times n},\ \mathbf{b}_2 \in R^{m_2} \\ \quad \mathbf{x} \geq \mathbf{0} & \mathbf{x} \in R^n \end{array}$$

To transform the general LP problem to any of these equivalent forms, elementary operations are used, such as:

* min $\mathbf{f(x)} = -\max[-\mathbf{f(x)}]$
* if x is arbitrary then $x = x^+ - x^-$, where $x^+ = \max[0, x]$ and $x^- = \max[0, -x]$,
* $\{\mathbf{a}^T\mathbf{x} = b, \mathbf{a} \in R^n, \mathbf{x} \in R^n\} \equiv \{\mathbf{a}^T\mathbf{x} \geq b \text{ and } \mathbf{a}^T\mathbf{x} \leq -b\}$,
* $\mathbf{a}^T\mathbf{x} \geq b$ may be replaced by $\mathbf{a}^T\mathbf{x} - x_s = b$, $x_s \geq 0$, x_s is called a slack variable.

1.4 Algorithms for Solving LP Problems

In general, two strategies for the solution of LP problems can be identified.

- An algorithm that traverses the boundary of the feasible region to locate the optimum extreme point ; these are called exterior methods.
- An algorithm that moves through the interior of the feasible region to arrive at the optimum extreme point; these are called interior methods.

Figure 1.1 illustrates how optimal solutions are approached by the two methods. In Figure 1.1 the progress of the interior method is shown by circles while that of the exterior method is shown by dots.

These approaches will now be described in more detail.

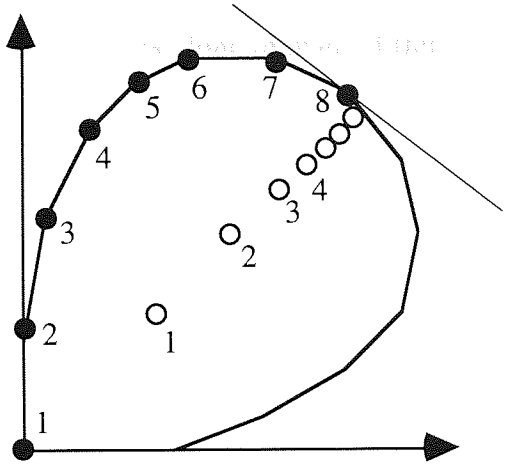

Figure 1.1 A Geometric Illustration of Interior and Exterior Methods.

1.5 The Simplex Method

The well-established, simplex algorithm, developed by Dantzig in the late 1940s, is an exterior method. After computing an initial, feasible solution (i.e., an extreme point), the simplex algorithm moves along the boundary from one extreme point to the next, always ensuring that the extreme point visited moves closer to the optimum solution. This is done by changing one of the vectors of the current basis with a non-basic vector, which becomes basic after pivoting. Thus, algebraically, moving from a vertex to an adjacent one corresponds to changing the current basis with an adjacent one. In a m×n LP problem, where $m < n$, a vertex is determined by m linearly independent tight constraints.

Two phases are generally required by the simplex method. In the first, the feasibility (or otherwise) of the problem is established and the vertex of the domain of the LP problem is found, if one exists. A monotone path of feasible points, in accordance with the objective function of the problem, is generated in the second phase of the algorithm. The path stops at a vertex when no improvement in the objective function value is possible, or else at an unbounded edge (in which case, the problem is unbounded).

There are many variants of the simplex and they can differ a great deal. The way the feasible starting point is obtained can be done in many different ways. The pivoting criteria, i.e., the criteria for choosing the entering variables into the basis, can also be totally different from one variant to another. However, they all generate a monotone path which ends at an optimal solution to the LP problem, if such an optimal solution exists.

1.6 Interior Point Algorithms

As shown throughout the preceding sections, the simplex method, being the main representative of exterior point methods, obtains the optimum solution by moving "cautiously" along edges of the solution space that connect adjacent corners or extreme points. This is because the optimum of a linear programming problem is always associated with an extreme or corner point of the solution space. In essence, the simplex method translates the geometric definition of the extreme point into an algebraic definition. Although in practice the simplex method has served well in solving very large problems, the computational basis of the technique, theoretically, can result in an exponential growth in the number of iterations needed to reach the optimum solution.

Attempts to produce a computationally efficient procedure that "cuts" across the interior of the solution space, rather than moving along the edges, were unsuccessful until 1984, when N. Karmarkar produced a polynomial-time algorithm. The effectiveness of the algorithm appears to be in the solution of extremely large and difficult linear programming problems, [Karmarkar & Ramakrishnan, (1991)]. Karmarkar's algorithm and any other methods that approach the optimum solution of the LP problem through the interior of the solution space are called interior-point methods.

Interior point methods for mathematical programming problems were introduced by Frisch (1955) and were developed as a tool for non-linear programming by Fiacco and McCormick (1968). While Fiacco and McCormick noted that their proposed methods could be applied in full measure to linear programming, neither they nor any other researchers at

the time seriously proposed that interior point methods would provide a viable alternative to the simplex method for actually solving linear programming problems, [Lustig, et al., (1994)].

1.7 The Projective Algorithm of Karmarkar

The algorithm of Karmarkar, [Karmarkar, (1984a); (1984b)], came as a result of the search for a method which has polynomial complexity like the ellipsoid method of Khachyan and the simplicial methods but is practical like the simplex. It is related to classical, interior point methods, but presents original features, such as the use of projective geometry and a logarithmic potential function to measure convergence.

Going in the direction of the gradient is the classical approach when interior point methods are considered for linear programming. However, this yields a substantial improvement in the objective function only if the current feasible point is at the centre of the polytope, i.e., sufficiently distant from all its boundaries. Consequently, for an iterative process to work with these ideas, it must alternate between centring the feasible point and taking a step in the gradient direction.

Classical interior methods of the Brown-Koopmans type have difficulties near the boundaries, precisely because they lack the centring step. The difficulties, usually, result in the loss of feasibility and slow convergence. On the other hand, Karmarkar's algorithm avoids the difficulties of the classical methods and successfully combines these two steps.

The initial paper of Karmarkar, [Karmarkar, (1984a); (1984b)], required the linear program to be expressed in the special homogeneous form

$$\begin{aligned} \text{Min} \quad & \mathbf{c}^T\mathbf{x} \\ \text{s. t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{0} \\ & \mathbf{e}^T\mathbf{x} = 1 \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{1.1}$$

and that the value of the objective function at the optimum point be equal to zero, i.e. $\mathbf{c^T x^*} = 0$, where $\mathbf{x^*}$ is the optimal solution vector of (1.1).

The algorithm begins with an initial feasible estimate $\mathbf{x^0}$ to the solution of (1.1) and moves to a new estimate $\mathbf{x^1}$ via the use of projective transformations. If we define

$$\mathbf{X_0} = \text{diag}(\mathbf{x^0}) \text{ and } T(\mathbf{x}) = \frac{\mathbf{X_0^{-1}x}}{\mathbf{e^T X_0^{-1} x}}$$

where $T(\mathbf{x})$ is the required projective transformation, then the point corresponding to $\mathbf{x^0}$ in the transformed space is $\frac{1}{n}\mathbf{e}$. Let vector δ be

$$\delta = -\gamma[\mathbf{I} - \mathbf{B^T(BB)^{-1}B}]\mathbf{X_0 c} \qquad (1.2)$$

where γ is a scalar step parameter and $\mathbf{B} = \begin{bmatrix} \mathbf{AX_0} \\ \mathbf{e^T} \end{bmatrix}$. A new point $\mathbf{x'}$ in the transformed space (x'-space) is defined by

$$\mathbf{x'} = \frac{1}{n}\mathbf{e} + \delta.$$

The corresponding point $\mathbf{x^1}$ in the original space (x-space) is obtained by the inverse projective transformation $T^{-1}(\mathbf{x'})$ defined by

$$\mathbf{x^1} = T^{-1}(\mathbf{x'}) = \frac{\mathbf{X_0 x'}}{\mathbf{e^T X_0 x'}}$$

It is shown that by appropriately choosing γ in (1.2), the algorithm converges to some point $\tilde{\mathbf{x}}$ with $\mathbf{c^T\tilde{x}} < 2^{-L}\mathbf{c^T x^0}$ in O(nL) iterations, where L is the length of the input data, [Karmarkar, (1984a); (1984b)].

In order to prove the polynomial complexity of the projective method, the potential logarithmic function was introduced by Karmarkar.

$$p(\mathbf{x}) = \ln \mathbf{c^T x} - \sum_{j=1}^{n} x_j.$$

This function is proved to be reduced by, at least, a constant amount in each iteration for a proper choice of γ in (1.2), [Karmarkar, (1984a); (1984b)].

As one can see from the above description, the centring process of the algorithm is performed by rescaling the feasible region at each iteration using a projective transformation. This results in approximating the optimization problem with a minimization over a sphere of known centre and radius $[\frac{1}{n}\mathbf{e}, \gamma]$. The minimization over a sphere is then solved by taking a step to its boundary along the projected gradient direction. The rescaling process combined with the step along the negative projected gradient is then repeated until optimality is achieved or the problem is recognized to be unbounded or infeasible. A sketch of the optimization process is as follows (Figure 1.2).

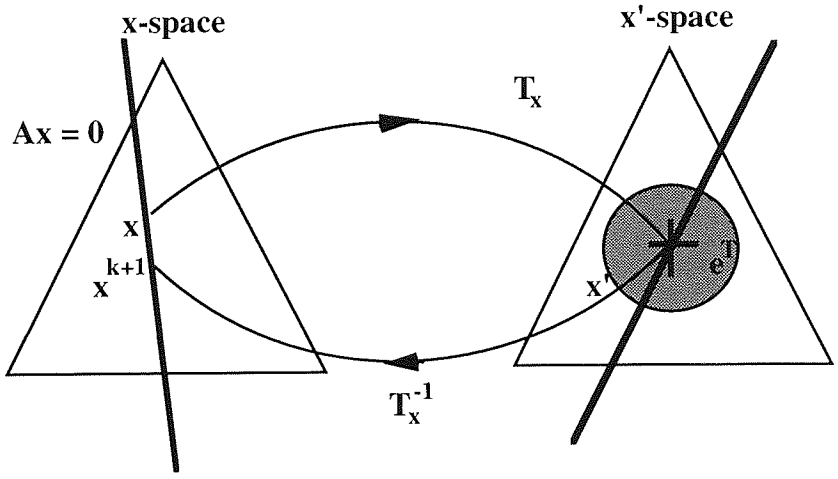

Figure 1.2 An Iteration of the Algorithm.

Although dual solutions were not generated in the original paper of Karmarkar, many researchers soon realized that the computation of the projected gradient in the main step of the algorithm provided values, in certain instances, [Fieldhouse & Tromans, (1985); Lustig, (1985)], which converge to the dual optimum solution, [Gay, (1987); Todd & Burrell, (1986); Ye & Kojima, (1987)].

The basic concept of duality is that every linear programming problem (called the primal) has an associated problem (called the dual). A solution to the dual is provided whenever a solution to the original primal problem is found. Thus, whenever a linear programming problem is solved, it actually provides the solution to two problems. The

primal-dual relationship is important in many respects. It is extensively used in the design of many variants of the simplex and, also, in the proof of theoretical results. In most implementations of interior-point algorithms, the duality gap, the difference between the value of the primal and the dual objective function, is used as a convergence criterion. Most large-scale implementations of algorithms based on the barrier-transformed problem associated with the primal LP, such as OB1, [Lustig, et al., (1991)] and ALPO, [Vanderbei, (1990)], reduce the barrier-penalty parameter as a function of the duality gap.

1.8 Using Barrier and Penalty Functions for Solving LP Problems

An important aspect of the Karmarkar algorithm is that of maintaining feasibility after each step and insuring reduction in the objective value monitored by the use of a logarithmic potential function. The idea is reminiscent of the barrier and penalty functions approach in non-linear programming due to Frish, (1955) and championed by Fiacco and McCormick, (1968).

Penalty function methods are characterized by their use of infeasible points. They are defined to ensure that iterates converge to a solution which is feasible. The squared Euclidean norm, usually attributed to Courant, (1943), leads to the well known quadratic penalty function:

$$\Phi(\mathbf{x}, \sigma) = \mathbf{f}(\mathbf{x}) + \frac{1}{2}\sigma \mathbf{c}(\mathbf{x})^{\mathbf{T}}\mathbf{c}(\mathbf{x})$$
$$= \mathbf{f}(\mathbf{x}) + \frac{1}{2}\sigma \|\mathbf{c}(\mathbf{x})\|_2^2$$

where the non-negative scalar σ is the penalty parameter. This was the earlier penalty function which provides points subject to $\mathbf{c}(\mathbf{x}) = \mathbf{0}$. The penalty is formed from a sum of squares of constraint violations while the σ determines the magnitude of the penalty.

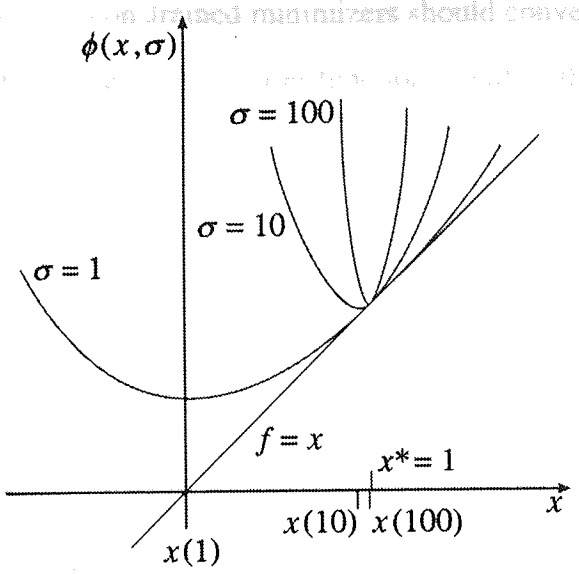

Figure 1.3 Penalty Functions

The above shows some graphs of $\phi(\mathbf{x}, \sigma)$ for the problem:

$$\begin{aligned} &\text{Min } x \\ &\text{s.t. } x - 1 = 0 \\ &\text{for which } P = x + \frac{1}{2}\sigma(x-1)^2 \end{aligned}$$

If the solution $x^* = 1$ is compared with the points which minimize $\phi(\mathbf{x}, \sigma)$, it is clear that $\mathbf{x}^*$ is a limit point of the penalty function as $\sigma \rightarrow \infty$. Hence, this suggests a technique of solving a sequence of minimization problems. It can be shown that $\mathbf{x}(\sigma^k) \rightarrow \mathbf{x}^*$ and that linear convergence is reached with one decimal place being obtained at each iteration. This behaviour can be justified for all problems.

Penalty functions, as well as barrier functions, suffer from a number of problems. Both are susceptible to ill-conditioning of the problem as the solution is approached. In addition, the convergence of penalty functions is dependent on the penalty parameter being sufficiently small, but it is difficult to determine the best choice from the information available.

In contrast to the penalty function methods, barrier functions are characterized by their ability to preserve strict constraint feasibility at all times by using a barrier term which is infinite on constraint boundaries. The barrier function creates a sequence of modified

functions whose successive, unconstrained minimizers should converge, in the limit, to the solution of the constrained problem. The barrier function modifies the objective function in such a way that successive iterates are kept 'inside' the feasible region. This is achieved by creating a 'barrier' at the boundary of the feasible region. The most popular barrier functions are the logarithmic barrier function, usually attributed to Frisch, (1955):

$$\textbf{Min } \mathbf{B(x,\ r)} = \mathbf{f(x)} - r \sum_{i=1}^{m} \mathbf{ln(c_i(x))}$$

and the inverse barrier function, [Carroll, (1961)]:

$$\textbf{Min } \mathbf{B(x,\ r)} = \mathbf{f(x)} + r \sum_{i=1}^{m} \frac{1}{\mathbf{c_i(x)}},$$

r, the barrier parameter, is the positive weight in both functions.

Function $\mathbf{B(x,\ r)}$ will never cross the barrier because as $\mathbf{c_i(x)} \rightarrow 0$ the second term of the barrier functions tends to infinity.

As with σ in the penalty function method, r is used to control the barrier function iteration. In this case, however, $r^k \rightarrow 0$ ensuring that the barrier term becomes negligible except close to the boundary. Vector $\mathbf{x}(r^k)$ is defined as the minimizer of $\mathbf{B(x,\ r)}$.

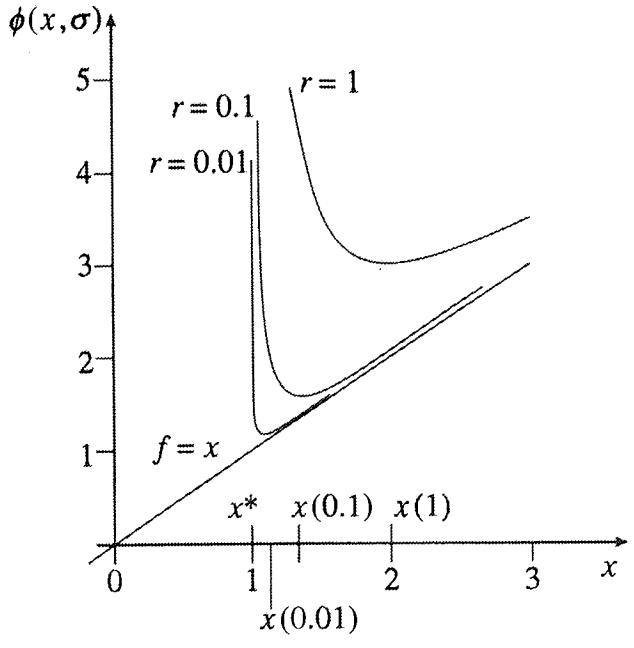

Figure 1.4 Barrier Functions

Figure 1.4 shows typical graphs for a sequence of values of r^k. It can be seen that $\mathbf{x}(r^k) \rightarrow \mathbf{x}^*$ as $r^k \rightarrow 0$, which can be established in a similar way to penalty function convergence.

Unfortunately, in addition to the problems due to ill-conditioning mentioned in the penalty functions case, other difficulties often arise. The barrier function is undefined for infeasible points, which can lead to line search being inefficient. Also, an initial feasible point is required, which is a non trivial problem in itself.

Gill, et al., (1985) suggested using the barrier function approach to LP and developed a class of projected Newton algorithms in which Karmarkar's algorithm can be shown to be a particular case, [Fletcher, (1986)]. LP problems are handled by being transformed into a non-linear programming problem of the form:

$$\textbf{BAP: Min} \quad \Phi(\mathbf{x}) = \mathbf{c}^T\mathbf{x} - \mu\sum_i \log x_i$$

$$\textbf{s.t.} \quad \mathbf{Ax} = \mathbf{b},$$

$\mu > 0$, μ is the barrier parameter.

The algorithm proceeds from a feasible point $\mathbf{x} > \mathbf{0}$ following the Newton direction $\mathbf{d} = \nabla^2\Phi(\mathbf{x}))^{-1}\nabla\Phi(\mathbf{x})$ projected onto Ker($\mathbf{A}$) to get a feasible point $\mathbf{y}$ (i.e. $\mathbf{Ay} = \mathbf{b}$). More explicitly, the Newton search direction $\mathbf{d}$ is obtained as the solution of the quadratic programming problem which is the minimization of a quadratic approximation of $\Phi(\mathbf{x})$ under feasibility constraints. This amounts to the problem:

$$\textbf{BQP: Min} \quad \nabla\Phi(\mathbf{x})\mathbf{d} + \frac{1}{2}\mathbf{d}^T\nabla^2\Phi(\mathbf{x})\mathbf{d}$$

$$\textbf{s.t.} \quad \mathbf{Ad} = \mathbf{0}$$

where $\nabla\Phi(\mathbf{x}) = \mathbf{c} - \mu\mathbf{D}^{-1}\mathbf{e}$ is the gradient of $\Phi(\mathbf{x})$, $\nabla^2\Phi(\mathbf{x}) = \mu\mathbf{D}^{-2}$ the Hessian, and $\mathbf{D} = \mathbf{diag}(x_1, x_2, ..., x_n)$, $\mathbf{x}$ being a feasible point to problem **BAP**.

The solution of **BQP** gives $\mathbf{d} = \mathbf{x} - \frac{1}{\mu}\mathbf{D}^2(\mathbf{c} - \mathbf{A}^T\lambda)$, where λ is the solution to the normal equations $\mathbf{AD}^2\mathbf{A}^T\lambda = \mathbf{AD}^2\mathbf{c}$, and the barrier parameter is chosen as $\mu = \mathbf{x}^T\mathbf{D}(\mathbf{c} - \mathbf{A}^T\lambda)$, [Fletcher, (1986)]. Vector $\mathbf{d}$ is a descent direction as the Hessian is positive definite when $\mathbf{x} > \mathbf{0}$. Thus, a step of length α along $\mathbf{d}$ results in point $\mathbf{y} = \mathbf{x} + \alpha\mathbf{d}$ such that, $\Phi(\mathbf{y}) < \Phi(\mathbf{x})$. Hence, an alternative process can be constructed. Gill, et al., (1985) showed that the projected Newton barrier method, for some parameter μ, generates a path parallel to

that followed by the projective algorithm. For $\mu = 0$ the barrier method is similar to the linear rescaling algorithm of Vanderbei, et al., (1986).

It has often been noted in the literature that the barrier parameter μ serves as a centring parameter, pulling the primal variables and dual slacks away from zero. Algorithms for choosing an initial μ^0 and reducing μ at each step in order to assure polynomial convergence of barrier methods were developed first for primal algorithms by Gonzaga, (1987), and applied to the primal-dual algorithm by Monteiro and Adler (1989). Since these algorithms reduce μ by a very small multiple at each step, they are hopelessly slow in practice.

In McShane, et. al., (1989), μ is chosen by a formula based on the duality gap and whether feasibility is obtained or not. Simply stated, μ serves principally as a feasibility parameter. Conceptually, the idea is that as μ is allowed to increase, the search vector points away from the boundary of the feasible region into the interior, and thus allows for greater step-lengths α before the nonnegativity constraints restrict the step.

1.9 Recent Developments in Interior Point Algorithms

The practical use of the original Karmarkar algorithm is made difficult by the assumptions required but, also, by the need for accurate computation and the use of a constant step-length throughout the algorithm. In later years, strategies which relax these assumptions were developed. Linear transformations, [Kortanek & Shi, (1987); Vanderbei, et al., (1986)], in other words different scaling, were also investigated. More classical interior point methods, such as Newton methods, [De Ghellinck & Vial, (1986)], and barrier methods, [Gill, et al., (1985)], which were originally intended for non-linear optimization were also developed. Finally, procedures for improving the rate of convergence of interior point methods (projective and affine scaling algorithms) for linear programming were proposed, [Kovacevic-Vujcic, (1991)]. For a very good survey of search directions used in interior point methods the reader is referred to Hertog and Ross, (1991).

A scaled potential algorithm was proposed by Anstreicher, (1989). An affine potential reduction algorithm that simultaneously seeks feasibility and optimality and that is closely related to that of Anstreicher was described by Todd, (1993). The new features of this algorithm are that a two-dimensional programming problem is used to derive better lower bounds and that the direction-finding subproblems used treat phase I and phase II of the algorithm more symmetrically. The above projective methods that combine phase I and phase II are among the methods discussed in the comparative paper of Todd and Wang, (1993). The super-linear and quadratic convergence theory of the duality gap sequence of these primal-dual interior-point methods is analysed and proved in Zhang and Tapia (1992).

An infeasible dual affine scaling method, which can be viewed as a dynamic "big-M" method, was proposed by Andersen, [Andersen, (1993)]. In contrast to the "big-M" method, this algorithm always finds a feasible point. This method, as well as the affine scaling algorithm of Barnes, [Barnes, (1986)], and a variant of the dual projective algorithm, [Karmarkar & Ramakrishnan, (1991)], will be described in detail in Chapter 4.

A primal-dual interior point algorithm for linear programming was introduced by Megiddo, (1986), who used logarithmic barrier methods to solve the primal and dual problems simultaneously. His method was first developed as an algorithm by Kojima, Mizuno, and Yoshise in 1989. The basic primal-dual logarithmic barrier method has been fully documented in McShane, et. al., (1989) and Choi, et. al., (1990). The algorithm eliminates the inequality constraints by incorporating them in the objective function, using the barrier approach. The search directions for both the next primal and dual iterates are produced using Newton's method applied on a system of linear equations. This system is the first order necessary conditions for the Lagrangian of the logarithmic barrier function. A detailed description of one of the latest implementations of the algorithm can be found in Lustig, et. al., (1991).

The theoretical efficiency of solving a standard-form LP by solving a sequence of shifted-barrier problems was examined in Freund, (1991). The advantage of using the shifted-barrier approach is that a starting feasible solution is unnecessary and there is no need for a phase-I–phase-II approach to solving the linear program, either directly or through the

addition of an artificial variable. Furthermore, the algorithm can be initiated with a "warm start", i.e., an initial guess of a primal solution that need not be feasible.

In Hertog, et al., (1992), the classical logarithmic barrier functions are combined with the use of the Newton method. Line searches are performed along the Newton direction with respect to the strictly convex logarithmic barrier function if the current estimate of the solution is far away from the central trajectory. If the current point is sufficiently close to this path, with respect to a certain metric, the barrier parameter is reduced.

For a survey of the most significant developments in the field of interior point methods the reader is referred to Lustig, et al., (1994) and Todd, (1994).

1.10 Research Needs and Objectives

Much of the work done in LP over the past thirty years has been concerned with improving existing simplex variants and developing new ones. It is only in the last decade that polynomial time algorithms became a topic of wide interest. This interest stems from LP being widely used on its own and as a building block in many optimization problems, such as structured problems with decomposable constraint set.

The overall objective of the present research is to investigate some aspects of the latest interior point methods, [Andersen, (1993); Barnes, (1986); Karmarkar & Ramakrishnan, (1991)], such as the preponderance of least square techniques in their efficient implementation and the use of preconditioners. Advanced preconditioning techniques were, therefore, called upon to cut down the number of iteration steps required to obtain a good approximation to the solution of the least square problem. A comparative study of these preconditioners based on extensive experiments of randomly generated problems was carried out and some conclusions about them were drawn. Sparsity is undoubtedly the important issue in any efficient implementation of these algorithms. To study the influence of sparsity over the performance of interior point algorithms the sparse option provided in

MATLAB was used. Experiments were carried out on randomly as well as on non-randomly generated LP problems for different levels of density of the constraint matrix.

Structured LP problems constitute an important class to which much work has been devoted in the frame of the simplex method leading to the design of elegant decomposition algorithms such as the Dantzig-Wolf algorithm, Rosen's partitioning algorithm, and others. However, these algorithms never outclassed interior point or simplex methods. Thus, it became necessary to consider the applicability of interior-point algorithms in conjunction with some new decomposition principles.

In order to implement some of the latest decomposition algorithms, it became clear there was a need to implement efficient quadratic programming solvers. Three algorithms, based on different principles each, were implemented and tested on a wide range of problems. For these tests randomly generated problems and problems obtained from text books were used. The influence of sparsity for both the constraint matrix and the cost vector over the performance of these three algorithms was also studied.

Chapter 2

Computation of the Projected Gradient

2.1 Introduction

After describing the latest work in the field of interior point methods, there remains one major problem to face for their efficient implementation, namely the computation of the search direction **p**. The present chapter, therefore, looks at the least squares (LSQ) problem and finds out what is available that can be used in the implementation of these algorithms.

Very few implementations discussed in Chapter 1 do not involve solving a LSQ problem when computing the projection matrix in the main step of the algorithm. It has been argued, [Tomlin, (1985)], that the efficiency of the projective algorithm is limited by the technology for solving LSQ problems. Efficient solution of the LSQ problem is, also, relevant to the implementation of other interior-point algorithms considered in Chapter 4 and to decomposition techniques considered in Chapter 6. Based on these arguments, it was necessary to review some important results considering the LSQ problems and investigate some techniques for improving their performance.

2.2 The Linear Least Squares Problem

The method of least squares is widely used in different fields of pure and experimental science that require the solution of a system of linear equations. Areas in which least squares arise include geodesy, photogrammetry, image enhancement, structural analysis, data smoothing, and mathematical programming. In numerical analysis, least

squares is used as an "extension" to the well-known Gaussian Elimination for non-square systems of linear equations.

The use of LSQ is credited to Gauss but there are references to it that go back a thousand years ago, [Longley, (1984)]. Methods for LSQ problems predate computers, although the development of efficient algorithms with sparsity and numerical stability considerations are recent and are strongly linked to the availability of digital computers.

2.2.1 LSQ Problem and Normal Equations

The LSQ problem is to minimize the norm of the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ of a system of linear equations

$$\mathbf{Ax} = \mathbf{b}. \qquad (2.2.1)$$

Although any norm may be used, it is generally the Euclidean norm which is considered. The LSQ problem is formulated as follows:

$$\min_x \|\mathbf{b} - \mathbf{Ax}\|^2 \qquad (2.2.2)$$

There is a strong relation between the LSQ problem and the normal equations. They are naturally derived as follows.

The residual vector $\mathbf{r}$ must be orthogonal to the column space of $\mathbf{A}$ (or space of $\mathbf{A}^T$). This condition is expressed as $\mathbf{A}^T\mathbf{r} = \mathbf{0}$ or $\mathbf{A}^T(\mathbf{b} - \mathbf{Ax}) = \mathbf{0}$, which leads to the normal equations $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$. The cross-product $\mathbf{A}^T\mathbf{A}$ is a positive definite matrix.

2.2.2 Data Characteristics and Algorithm Performance

Many techniques are available to solve the LSQ problem. A successful method for solving the LSQ problem should take into account the special characteristics these problems may have.

The ill-conditioning of the matrix **A** is one of the crucial characteristics of a LSQ problem because large-scale and ill-conditioned problems are difficult and expensive to solve. Continuous research on that subject concludes that scaling and preconditioning of the data are the most powerful methods for the solution of these problems. Despite the fact that there is no standard way for scaling or preconditioning, methods achieve numerical stability when these techniques are successfully used.

Sparsity is another characteristic that makes techniques for LSQ differ in their numerical properties and execution time. When sparse matrices are considered special preconditioners which preserve the sparsity of the original matrix are recommended. Numerous heuristics based usually on reordering are available for the exploitation of sparsity. In subsequent chapters, the relevance of sparsity considerations for the implementation of interior point algorithms will be underlined.

2.2.3 Numerical Stability and Condition Number

Errors always occur as a result of an operation performed during computation with finite precision. In subsequent calculations this error is usually increased and there are cases in which errors grow so large that the computed result is totally inaccurate. A procedure leading to such results is labelled numerically unstable. However, some problems are inherently unstable or ill-conditioned, which may cause most of the procedures on such problems to perform badly regardless of the precautions taken, [Kronsjö, (1987)].

To measure the instability of a solution and ill-conditioning of the corresponding system of linear equations $\mathbf{Ax} = \mathbf{b}$, where **A** is square and non-singular, the term condition

number of the matrix **A**, k(**A**), is used. The condition number of a matrix is given by the expression $\|\mathbf{A}\|.\|\mathbf{A}^{-1}\|$, where $\|\cdot\|$ denotes the norm. As $\|\mathbf{I}\| = 1$, for every subordinate norm, and $\mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$ then $1 = \|\mathbf{I}\| \leq \|\mathbf{A}\|.\|\mathbf{A}^{-1}\| = k(\mathbf{A})$. Thus $k(\mathbf{A}) \geq 1$ for any matrix.

The condition number is a precise measure of linear system sensitivity and indicates the maximum effect of perturbations in **A** and **b** on the exact solution of $\mathbf{Ax} = \mathbf{b}$. If k(**A**) is "large", the exact solution may be substantially altered by even small changes in the data. This is because the relative error in **x** can be k(**A**) times the relative error in **A** and **b**. **A** is often said to be ill-conditioned or well-conditioned when k(**A**) is "large" or "small" respectively.

The condition number of a matrix can be obtained in MATLAB with the function COND. This function uses the singular value decomposition algorithm and defines the condition number as the ratio between the largest and the smallest of the singular values of the matrix, [Lindfield & Penny, (1995)].

In the chapter that follows the effect of the condition number on the convergence of the conjugate gradient method is studied. This is achieved with the use of the SPRANDSYM procedure of MATLAB that generates symmetric sparse matrices of random entries and given condition number.

2.2.4 Scaling and Preconditioning

The manipulation of a matrix in such a way that all entries are about 1 is called scaling. Scaling is not always feasible and there is no automatic way that satisfactorily scales any matrix. Scaling is usually done by multiplying the rows or columns of a matrix by a constant and aims to make the variables of the scaled problem have the same magnitude and order unity in the solution region. When scaling is used the scaling factors must be stored and be used to restore the original scaling before the final results are obtained.

By preconditioning of the system $\mathbf{Ax} = \mathbf{b}$ one means the multiplication of both the sides of the equation by a suitable matrix **Pre** that will create an equivalent system $\mathbf{A'y} =$

b', where **A' = Pre*A** and **b' = Pre*b**. Matrix **Pre** is chosen in such a way that the new system is easier to solve than the original one. Preconditioning can be viewed as a way of scaling which improves the condition of a matrix and a technique to accelerate convergence. Among the techniques for LSQ, the iterative ones are the most dependent on preconditioning, [Gill, et al., (1981)]. The use of several preconditioners with one of the most popular method for solving the LSQ problem, namely the conjugate gradient method, will be discussed in the next chapter.

2.2.5 Sparsity

Sparsity is often an important characteristic of large-scale matrices. A big percentage of the entries of a sparse matrix are zero. Sparse matrixes arise in many problems of science and engineering. Although it is difficult to exactly define a sparse matrix, a matrix is called sparse when it is profitable to exploit its zeros, [George & Liu, (1981)]. Exploiting the zero elements of a matrix is justified if sparsity is of such an extent that this feature can be practically utilized to reduce the computational time and storage facilities required for operations used on such matrices, [Lindfield & Penny, (1995)]

By exploiting the sparsity of a matrix one reduces the storage requirements of procedures for matrix computations. The actual technical details involving memory locations can be found in Tewarson, (1973) and De Buchet, (1971). Matrix operations involving sparse matrices are also less CPU time consuming. This is justified by the redundancy of the following operations.

If a is non-zero, then: $0 \cdot a = 0$, $0 + a = a$, $0 / a = 0$, $0 \cdot 0 = 0$, $0 + 0 = 0$.

In operations involving sparse matrices the above calculations are not performed as the results are known in advance. The big percentage of zero entries in a sparse matrix allows savings in storage space since only the non-zero entries are required in forming the product of **A** with an arbitrary vector or matrix of appropriate dimension. In other words, the non-

zero elements of a sparse matrix **A** are not stored explicitly but they are reproduced every time the matrix is used.

Using sparse data structure a matrix is represented in space proportional to the number of non-zeros entries only instead of the total number of the elements of the matrix. Sparse data structure effects also in matrix operations to compute results in time proportional to the number of arithmetic operations on non-zeros, [Gilbert, et al., (1992)]

One of the main problems in solving sparse systems is that when the matrix is factored, it suffers fill-in, i.e., non-zeros are created as a consequence of the factorization. Thus, sparsity tends to be destroyed. In the case of normal equations, for example, the Cholesky factor **L** has more non-zeros than the lower part of $\mathbf{A^TA}$. However, it has been observed that a judicious reordering of the matrix rows and columns can dramatically reduce fill-in. Such a reordering is practically embodied in a permutation matrix, which is defined as follows.

A permutation matrix **P** is a square matrix whose columns are some permutation of those of the identity matrix. Matrix **P** is orthogonal, i.e., $\mathbf{P^{-1} = P^T}$ and $\mathbf{P^TP = I}$.

To preserve the desirable characteristic of symmetry in a matrix only data reorderings of the form $\mathbf{PAP^T}$ are considered. Row permutations ($\mathbf{A \leftarrow PA}$) or column permutations ($\mathbf{A \leftarrow AP}$) alone destroy symmetry. A permutation update of the form ($\mathbf{A \leftarrow APA}$) is called a symmetric permutation of **A**. Symmetric permutations do not move off-diagonal elements to the diagonal, [Golub & Van Loan, (1983)].

The software used for the implementation of all the algorithms described in the following chapters, MATLAB ver. 4.2, allows computations with matrices in both dense and sparse formats. Dense matrices are the default option. Sparse matrices are declared with the use of the SPARSE function. Special sparse matrices like the identity matrix and matrices of given density can also be generated in MATLAB. Using the SPARSE option of MATLAB, the effect that sparse matrices have on the execution time of algorithms for linear and quadratic programming was studied in the following chapters.

Sparse matrix technology was founded by Ralph Willoughby of I.B.M. in the 1960's, [Duff, (1986)]. Since then, it has dominated the design of efficient software in numerical computations of large systems.

2.2.6 Solving the Least Square Problem

As mentioned earlier, there are many techniques for solving LSQ problems. The choice of a technique may be determined by two main criteria: numerical stability and sparsity exploitation (i.e., cost). Unfortunately, no single technique completely fulfils these criteria, as problems differ widely in the condition of their data and their sizes. For small scale problems, even when they are ill-conditioned, most techniques can be successfully applied. However, when the problems are large, the choice of a suitable technique becomes crucial.

Techniques for LSQ problems may be divided into two main categories:

- Direct
- Iterative

2.2.6.1 Direct Methods

a) Cholesky Factorization Technique

The Cholesky method is a technique to solve the system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a symmetric positive definite (SPD) matrix. A triangular factorization of $\mathbf{A}$ is obtained such that $\mathbf{A} = \mathbf{L}^T\mathbf{L}$, where $\mathbf{L}$ is a lower triangular matrix (see Appendix A for computation details of $\mathbf{L}$). Because of the symmetric positive definite characteristic of $\mathbf{A}$, the above factorization exists and moreover is stable to compute, [Golub & Van Loan, (1983)]. The system at hand may be written as:

$$\mathbf{L^T L x = b.}$$

Put $\mathbf{Lx = y,}$

and solve $\mathbf{L^T y = b}$

by a forward substitution. Then solve $\mathbf{Lx = y}$ by a backward substitution to obtain $\mathbf{x}$.

The algorithmic form of the Cholesky method applied to $\mathbf{Ax = b}$, with ordering brought into consideration, is given below, [Heath, (1984)].

Algorithm 2.0

Step 1. Find a permutation matrix $\mathbf{P}$.

Step 2. Factorize $\mathbf{P^T A^T A P}$ to find a sparse Cholesky factor $\mathbf{L}$.

Step 3. Solve $\mathbf{L^T z = P^T A^T b}$.

Step 4. Solve $\mathbf{Ly = z}$.

Step 5. Restore original order: $\mathbf{x = Py}$.

Speed is the main advantage of the Cholesky method, [Saunders, (1994)]. On the other hand, forming the cross-product $\mathbf{A^T A}$ destroys the sparsity of the original problem and produces a matrix which condition number is the square of that of $\mathbf{A}$. The latter is crucial for problems in which $\mathbf{A}$ is already poorly conditioned. Accurate solution to such LSQ problems may be difficult to achieve, if not impossible.

b) Orthogonal Methods

An orthogonal matrix $\mathbf{Q}$ is one which satisfies the relation $\mathbf{Q^T Q = I}$. Using orthogonal matrices the Cholesky factor $\mathbf{R}$ of a matrix $\mathbf{A}$ can be computed avoiding the explicit formation of the cross-product $\mathbf{A^T A}$. Matrices $\mathbf{A}$ and $\mathbf{b}$ are reduced into the following forms:

$$\mathbf{QA} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} \text{ and } \mathbf{Qb} = \begin{bmatrix} \mathbf{c} \\ \mathbf{d} \end{bmatrix}, \qquad (2.2.3)$$

where $\mathbf{c}$ is of order n, and $\mathbf{d}$ of order (m–n) and $\mathbf{R}$ is triangular (n×n)-matrix.

Based on the property of **Q**, it can be written

$$\mathbf{A}^T\mathbf{A} = \mathbf{A}^T\mathbf{I}\mathbf{A} = \mathbf{A}^T\mathbf{Q}^T\mathbf{Q}\mathbf{A} = \begin{bmatrix}\mathbf{R}^T & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{R} \\ \mathbf{0}\end{bmatrix} = \mathbf{R}^T\mathbf{R}.$$

This shows that **R** is the Cholesky factor of $\mathbf{A}^T\mathbf{A}$.

Three main methods are available for computing the reductions (2.2.3):

- Gram-Schmidt Orthogonalization, [Longley, (1984)]
- Givens Rotations, [Golub & Van Loan, (1983)]
- Householder Reflections, [Kronsjö, (1987)]

The main disadvantage of these three methods is that they do not preserve sparsity. In other words, even if **A** and **R** are sparse, it is unlikely that the orthogonal matrix **Q** will be particularly sparse.

2.2.6.2 Iterative Methods

In many situations iterative methods may be preferred over direct methods for one or more reasons. Iterative methods may be good alternatives to direct methods for some large-scale sparse LSQ problems. One of their advantages (and that of all iterative processes for any class of problems, like the projective algorithm itself) is the possibility of stopping the iterative process when an approximate solution to the problem at hand is reached. This, obviously, is not possible with direct methods. Another advantage is also the difficulty of obtaining an accurate solution with direct methods for some problems. In this respect, iterative methods are more suitable, as accuracy may be monitored. Finally, iterative methods usually need less memory requirements and are in general easier to program and to adapt to different types of problems, [Van der Vorst & Dekker, (1988)].

Consider the system of N linear equations (2.2.1), where **b** is a known vector, **x** is the unknown solution, and **A** is a (usually sparse) $N \times N$ matrix. An iterative method generates, given an initial approximation $\mathbf{x}^{(0)}$ to **x**, a sequence $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$,..., which will

hopefully converge to the solution $\mathbf{x}$. It can be assumed that $\mathbf{x}^{(0)}$ is the zero-vector. This is not a restriction, as (2.2.1) could be rewritten as

$$\mathbf{A}\ (\ \mathbf{x} - \mathbf{x}^{(0)}\ ) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)} \qquad (2.2.4)$$

and instead of (2.2.1) the solution of (2.2.4) might be considered.

A simple iterative technique consists of a *Richardson-iteration* , Varga, (1962)

$$\mathbf{x}^{(n+1)} = (\ \mathbf{I} - \mathbf{A}\ )\ \mathbf{x}^{(n)} + \mathbf{b}, \qquad (2.2.5)$$

from which it is easily seen that $\mathbf{x}^{(n+1)}$ is a linear combination of the vectors: $\mathbf{b}$, $\mathbf{Ab}$,..., $\mathbf{A}^n\mathbf{b}$

Definition 2.1 The space spanned by the vectors $\mathbf{b}$, $\mathbf{Ab}$,..., $\mathbf{A}^n\mathbf{b}$ is called the Krylov subspace $\mathbf{K}^{(n+1)}(\mathbf{A};\mathbf{b})$.

Most iterative methods have in common that they select as iterates vectors from the Krylov space $\mathbf{K}^{(n)}(\mathbf{A};\mathbf{b})$ for n = 1, 2, ..., but they differ in the choice of the selection criteria. For example, one could choose the vector $\mathbf{x}^{(n+1)} \in \mathbf{K}^{(n+1)}(\mathbf{A};\mathbf{b})$ such that the residual

$$\mathbf{r}^{(n+1)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(n+1)}$$

is minimized in Euclidean norm. The drawback is, however, that the computation of the residual requires an additional matrix-vector multiplication. For some special class of systems, this drawback can be circumvented by minimizing in a different norm.

a) Conjugate Gradient Methods

Conjugate gradient methods are popular because of their robustness and stability for large problems. They are called upon to replace direct methods, when these are not viable because of the size or density of the problems matrices. Conjugate gradient methods refer to a wide class of optimization algorithms which generate search directions without storing a

matrix, [Gill, et al., (1984)]. There are two types of conjugate gradient methods: the linear and the non-linear methods. For the purpose of this research, linear conjugate gradient methods were considered.

Originally, conjugate gradient methods were designed to solve, iteratively, positive definite systems of linear equations. The iterative process uses the relation $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k$, where α_k is a non-negative scalar called step-length, and $\mathbf{p}_k$ a vector direction of search . The vector $\mathbf{p}_k$ is obtained as follows:

If the positive definite system to be solved is $\mathbf{Qx} = -\mathbf{c}$, the direction of search can be computed as $\mathbf{p}_{k+1} = -(\mathbf{Q}\mathbf{x}^{(k+1)} + \mathbf{c}) + \beta_k \mathbf{p}_k$, with

$$\beta_k = \frac{\mathbf{g}_{k+1}^T \mathbf{Q} \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{Q} \mathbf{p}_k}$$

where $\mathbf{g}_k = \mathbf{Q}\mathbf{x}^{(k)} + \mathbf{c}$. The step-length α_k is evaluated with the formula

$$\alpha_k = - \frac{\mathbf{g}_{k+1}^T \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{Q} \mathbf{p}_k}$$

In general, if one supposes that $\mathbf{A}$ is positive definite then its inverse $\mathbf{A}^{-1}$ is also positive definite. The minimization of the residual in the norm $\|\cdot\|_{A^{-1}}$, where $\|\mathbf{x}\|_A \equiv \sqrt{(\mathbf{Ax},\mathbf{x})}$, leads to the conjugate gradient, [Golub & Van Loan, (1983)], i.e. the iterate $\mathbf{x}^{(n+1)}$ satisfies

$$\| \mathbf{b} - \mathbf{A}\mathbf{x}^{(n+1)} \|_{\mathbf{A}^{-1}} \leq \| \mathbf{b} - \mathbf{A}\mathbf{y} \|_{\mathbf{A}^{-1}}, \forall\ \mathbf{y} \in \mathbf{K}^{(k+1)}\ (\mathbf{A};\mathbf{b}),$$

or equivalently,

$$\| \mathbf{x}^{(n+1)} - \mathbf{x} \|_{\mathbf{A}} \leq \| \mathbf{y} - \mathbf{x} \|_{\mathbf{A}}, \forall\ \mathbf{y} \in \mathbf{K}^{(k+1)}\ (\mathbf{A};\mathbf{b}).$$

The rate of convergence of the conjugate gradient method is known to be dependent on the distribution of the eigenvalues of the matrix $\mathbf{A}$. Let λ_{max} and λ_{min} be the largest and smallest eigenvalue of the positive definite matrix $\mathbf{A}$. Then, one would have ,approximately,

$$\| \mathbf{r}^{(k)} \| \approx \left(1 - 2\sqrt{\lambda_{min} / \lambda_{max}} \right)^k \| \mathbf{r}^{(0)} \| .$$

When the smallest (largest) eigenvalue lies isolated, the rate of convergence improves during the iteration process, [Van der Sluis & Van der Vorst, (1986)]. The condition number **Cond(A)** of **A**, which equals $\lambda_{max} / \lambda_{min}$, should, in general, be small in order to obtain fast convergence. However, in many applications, e.g., the discretized Poisson equation, **A** has a very large condition number. Consequently, it is important to modify equation (2.2.4), multiplying with a suitable preconditioner $\mathbf{Q^{-1}}$, and solve the equation

$$\mathbf{Q^{-1}Ax = Q^{-1}b}$$

instead. The condition number of $\mathbf{Q^{-1}A}$ may be considerably less than $\lambda_{max} / \lambda_{min}$, resulting in a significantly decreased number of iteration steps, at the cost of some additional overhead in constructing and multiplying with $\mathbf{Q^{-1}}$. Later in this chapter a survey of several preconditioners is given which may be of value for the equations that require solving.

The conjugate gradient method is not suitable for non-symmetric problems, therefore, alternative methods that may be used in this case will be discussed next.

b) Solving the Normal Equations

One way to get around the difficulties caused by the unsymmetry of **A** consists in first deriving the normal equation from (2.2.1)

$$\mathbf{A^TAx = A^Tb} \qquad (2.2.6)$$

and then solving this positive definite system using the conjugate gradient method. However, the condition number of $\mathbf{A^TA}$ is the square of the condition number of **A**, so one might expect slow convergence and, in particular for ill-conditioned systems, round-off errors may contaminate the result. The latter disadvantage is avoided in the LSQR method, which is equivalent to (2.2.6) in exact arithmetic.

c) LSQR Algorithm of Paige & Saunders

LSQR algorithm was designed to solve non-symmetrical systems of linear equations, LSQ problems, and damped LSQ problems of the form:

$$\min_{\lambda} \left\| \begin{pmatrix} \mathbf{A} \\ \lambda \mathbf{I} \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{b} \\ 0 \end{pmatrix} \right\|_2 \qquad (2.2.7)$$

where λ is a scalar. The algorithm was intended to solve large and sparse problems. It is based on the algorithm of Golub and Kahan, cited by Paige and Saunders, (1982), to reduce matrix **A** to a lower diagonal form. However, this algorithm is itself a variant of the Lanczos process (or tridiagonalization) for symmetric matrices. The solution to (2.2.7) satisfies the symmetric system

$$\begin{bmatrix} \mathbf{I} & \mathbf{A} \\ \mathbf{A}^T & -\lambda^2 \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}, \qquad (2.2.8)$$

where $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$. Note that (2.2.8) is a symmetric system. Hence, application of the Lanczos process is possible and leads to the forms

$$\begin{bmatrix} \mathbf{I} & \mathbf{B_k} \\ \mathbf{B_k}^T & -\lambda^2 \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{t_{k+1}} \\ \mathbf{y_k} \end{bmatrix} = \begin{bmatrix} \beta_1 \mathbf{e_1} \\ \mathbf{0} \end{bmatrix},$$

$$\begin{bmatrix} \mathbf{r_k} \\ \mathbf{x_k} \end{bmatrix} = \begin{bmatrix} \mathbf{U_{k+1}} & \mathbf{0} \\ \mathbf{0} & \mathbf{V_k} \end{bmatrix} - \begin{bmatrix} \mathbf{t_{k+1}} \\ \mathbf{y_k} \end{bmatrix},$$

where $\mathbf{B_k}$ is (k+1)×k and lower bidiagonal and $\mathbf{y_k}$ is the solution of the damped least square problem

$$\min_{\mathbf{y_k}} \left\| \begin{pmatrix} \mathbf{B_k} \\ \lambda \mathbf{I} \end{pmatrix} \mathbf{y_k} - \begin{pmatrix} \beta_1 \\ 0 \end{pmatrix} \right\|_2$$

Orthogonal transformations may then be used to reliably solve it.

The algorithm LSQR is analytically equivalent to conjugate gradient methods. It generates a sequence of approximations $\{\mathbf{x_k}\}$ such that the residual norm $\|\mathbf{r_k}\|$ is monotonically reduced. Paige and Saunders, (1982) claim that it is numerically more reliable than the standard conjugate gradient methods, in various circumstances. The costs of the method per iteration are nevertheless increased by a factor of two in comparison with an iteration for (2.2.1), as two matrix-vector multiplications are now necessary.

Another disadvantage of the last two described methods can be that a preconditioning for (2.2.6) and (2.2.7) could be more cumbersome than the one for (2.2.1) as these systems

are less sparse, especially in the case of (2.2.6). However, the methods are guaranteed to converge anyhow (even in a finite number of iterations, neglecting round-off), so they might be of value in situations where other methods that are cheaper per iteration step fail.

Chapter 3

Conjugate Gradient Methods and Preconditionining

3.1 Introduction

The computationally intensive step of most interior point algorithms is the solution of a system of linear equations

$$\mathbf{Ax} = \mathbf{b} \qquad (3.1)$$

In the above system **A** is usually a symmetric and positive definite matrix of the form $\mathbf{ED^2E^T}$. Although implementations of direct methods, such as **QR** factorization of $\mathbf{DE^T}$ and Cholesky factorization of $\mathbf{ED^2E^T}$, can give good speed-ups on many real-world problems, very large speed-up factors on very large problems require a specialized implementation of the preconditioned conjugate gradient method, [Karmarkar & Ramakrishnan, (1991); Ponnambalam, et al., (1992)].

In general, conjugate direction methods can be regarded as being intermediate between the method of steepest descent and Newton's method. They are motivated by the desire to accelerate the typically slow convergence associated with steepest descent while avoiding the information requirements associated with the evaluation, storage, and inversion of the Hessian (or at least solution of a corresponding system of equations) as required by Newton's method.

In this chapter preconditioning techniques will be examined as a way to accelerate the performance of the conjugate gradient algorithm for the solution of systems of linear equations .

3.2 Conjugate Directions

Definition. Given a symmetric matrix $\mathbf{A}$, two vectors $\mathbf{d_1}$ and $\mathbf{d_2}$ are said to be $\mathbf{A}$-orthogonal, or conjugate, with respect to $\mathbf{A}$, if $\mathbf{d_1^T A d_2} = 0$.

In the applications considered for this research the matrix $\mathbf{A}$ will be positive definite, but this is not inherent in the basic definition. Thus, if $\mathbf{A} = \mathbf{0}$, any two vectors are conjugate, while if $\mathbf{A} = \mathbf{I}$, conjugacy is equivalent to the usual notion of orthogonality. A finite set of vectors $\mathbf{d_0}, \mathbf{d_1},..., \mathbf{d_k}$ is said to be a $\mathbf{A}$-orthogonal set if $\mathbf{d_i^T A d_j} = 0$ for all $i \neq j$.

Proposition. If $\mathbf{A}$ is positive definite and the set of non-zero vectors $\mathbf{d_0}, \mathbf{d_1},..., \mathbf{d_k}$ are $\mathbf{A}$-orthogonal, then these vectors are linearly independent.

Proof. Suppose there are constants α_i, i = 0, 1, 2,..., k such that $\alpha_0\mathbf{d_0} + ... + \alpha_k\mathbf{d_k} = \mathbf{0}$. Multiplying by $\mathbf{A}$ and taking the scalar product with $\mathbf{d_i}$ yields $\alpha_i\mathbf{d_i^T A d_i} = 0$. Or, since $\mathbf{d_i^T A d_i} > 0$ in view of the positive definiteness of $\mathbf{A}$, the result is $\alpha_i = 0$.

3.3 The Conjugate Gradient Method

The conjugate gradient method (CGM) is obtained by selecting the successive direction vector as a vector conjugate to the preceding vector for each step of the method. Thus, the directions are not specified beforehand, but rather are determined sequentially at each step of the iteration. At step k one evaluates the current negative gradient vector and adds to it a linear combination of the previous direction vector to obtain a new conjugate direction vector along which to move. There are two primary advantages to this method of direction selection.

First, until the solution is reached, the gradient is always non-zero and linearly independent of all previous direction vectors. Indeed, the gradient is orthogonal to the

subspace generated by the directions $\mathbf{d_0}, \mathbf{d_1},..., \mathbf{d_{k-1}}$. When the solution is reached the gradient vanishes and the process terminates.

Second, an important advantage of the conjugate gradient method is the especially simple formula that is used to determine the new direction vector.

Finally, the conjugate gradient algorithm, when applied to a positive definite quadratic problem, is guaranteed to converge in n or less steps. Since optimizing such a problem is equivalent in solving a system of equations one can expect the same convergence for the system $\mathbf{Ax = b}$, if $\mathbf{A}$ is not ill-conditioned.

3.4 The Conjugate Gradient Algorithm

The conjugate gradient algorithm as an optimization process can be described as follows:

Step 0. Set (a) $\mathbf{x_0} = 0$; (b) $\mathbf{r_0} = \mathbf{b}$; (c) $k = 0$
Step 1. **While** $\mathbf{r_k} \neq \mathbf{0}$
(a) $k = k + 1$
(b) **if** $k = 1$ **then**
$\mathbf{p_1} = \mathbf{r_0}$
else
$\beta_k = \mathbf{r}_{k-1}^T \mathbf{r}_{k-1} / \mathbf{r}_{k-2}^T \mathbf{r}_{k-2}$
$\mathbf{p_k} = \mathbf{r_{k-1}} + \beta_k \mathbf{p_{k-1}}$
end
(c) $a_k = \mathbf{r}_{k-1}^T \mathbf{r}_{k-1} / \mathbf{p}_k^T \mathbf{A p_k}$
(d) $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_k \mathbf{p_k}$
(e) $\mathbf{r_k} = \mathbf{r_{k-1}} - \alpha_k \mathbf{A p_k}$
end, Step 2. $\mathbf{x} = \mathbf{x_k}$

Each succeeding step moves in a direction that is a linear combination of the current gradient and the preceding direction vector. The attractive feature of the algorithm is the simple formulae used for updating the direction vector. The conjugate gradient method requires only matrix-vector products to obtain the solution of a linear system of equations whose matrix is symmetric and positive definite.

3.5 Preconditioned Conjugate Gradient Algorithm

It has been proved, [Golub & Van Loan, (1983)], that the conjugate gradient method performs well on a system of linear equations $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is near the identity, either in the sense of a low rank perturbation or in the sense of norm. These properties can be expressed as $\mathbf{A} = \mathbf{I} + \mathbf{B}$, rank($\mathbf{B}$) = s or norm($\mathbf{A} - \mathbf{I}$) < t, , where s and t are small numbers. In this section, it is shown how to precondition a linear system so that the matrix of coefficients assumes one of the above "nice" forms.

Since $\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}\mathbf{C}\mathbf{x} = \mathbf{C}^{-1}\mathbf{b}$, i.e. $\bar{\mathbf{A}}\bar{\mathbf{x}}=\bar{\mathbf{b}} \equiv \mathbf{Ax}=\mathbf{b}$, applying the algorithm to the system $\bar{\mathbf{A}}\bar{\mathbf{x}} = \bar{\mathbf{b}}$, where $\bar{\mathbf{A}} = \mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}$, $\bar{\mathbf{x}} = \mathbf{C}\mathbf{x}$, $\bar{\mathbf{b}} = \mathbf{C}^{-1}\mathbf{b}$ and $\mathbf{C}$ is symmetric positive definite, as described in Golub and Van Loan, (1983), the following iteration is obtained.

Step 0. Set (a) k = 0; (b) $\bar{\mathbf{x}}_0 = \mathbf{0}$; (c) $\bar{\mathbf{r}}_0 = \bar{\mathbf{b}}$

Step 1. **While** $\bar{\mathbf{r}}_k \neq \mathbf{0}$

(a) k = k + 1

(b) **if** k = 1

$\bar{\mathbf{p}}_1 = \bar{\mathbf{r}}_0$

else

$\beta_k = \bar{\mathbf{r}}_{k-1}^T\bar{\mathbf{r}}_{k-1} \,/\, \bar{\mathbf{r}}_{k-2}^T\bar{\mathbf{r}}_{k-2}$

$\bar{\mathbf{p}}_k = \bar{\mathbf{r}}_{k-1} + \beta_k\bar{\mathbf{p}}_{k-1}$

end

(c) $\alpha_k = \bar{\mathbf{r}}_{k-1}^T\bar{\mathbf{r}}_{k-1} \,/\, \bar{\mathbf{p}}_k^T\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}\bar{\mathbf{p}}_k$

(d) $\bar{\mathbf{x}}_k = \bar{\mathbf{x}}_{k-1} + \alpha_k\bar{\mathbf{p}}_k$

(e) $\bar{\mathbf{r}}_k = \bar{\mathbf{r}}_{k-1} - \alpha_k\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}\bar{\mathbf{p}}_k$

end, Step 2. $\bar{\mathbf{x}} = \bar{\mathbf{x}}_k$

In view of the remarks in § 2.2.6.2, $\mathbf{C}$ should be chosen so that $\bar{\mathbf{A}}$ is well conditioned or a matrix with clustered eigenvalues. The latter means that all eigenvalues lie in a small range and the quotient of the maximum and the minimum eigenvalue is small, approaching one. In the above algorithm $\bar{\mathbf{x}}_k$ should be regarded as an approximation to $\bar{\mathbf{x}}$ and $\bar{\mathbf{r}}_k$ is the residual in the transformed coordinates, i.e., $\bar{\mathbf{r}}_k = \bar{\mathbf{b}} - \bar{\mathbf{A}}\bar{\mathbf{x}}_k$. Of course, once $\bar{\mathbf{x}}$ is determined, then $\mathbf{x}$ is obtained via the equation $\mathbf{x} = \mathbf{C}^{-1}\bar{\mathbf{x}}$. However, it is possible to avoid explicit reference to the matrix $\mathbf{C}^{-1}$ by defining $\bar{\mathbf{p}}_k = \mathbf{C}\mathbf{p}^k$, $\bar{\mathbf{x}}_k = \mathbf{C}\mathbf{x}_k$ and $\bar{\mathbf{r}}_k = \mathbf{C}^{-1}\mathbf{r}^k$. To demonstrate, these

three definitions are substituted into the above algorithm. Recalling that $\tilde{\mathbf{b}} = \mathbf{C}^{-1}\mathbf{b}$ and $\tilde{\mathbf{x}} = \mathbf{C}\mathbf{x}$, the following is obtained

Step 0. Set (a) k = 0; (b) $\mathbf{C}\mathbf{x}_0$; (c) $\mathbf{C}^{-1}\mathbf{r}_0 = \mathbf{C}^{-1}\mathbf{b}$
Step 1.**While** $\mathbf{C}^{-1}\mathbf{r}_k \neq \mathbf{0}$
(a) k = k + 1
(b) **if** k = 1
$\mathbf{C}\mathbf{p}_1 = \mathbf{C}^{-1}\mathbf{r}_0$
else
$\beta_k = (\mathbf{C}^{-1}\mathbf{r}_{k-1})^T(\mathbf{C}^{-1}\mathbf{r}_{k-1}) \,/\, (\mathbf{C}^{-1}\mathbf{r}_{k-2})^T(\mathbf{C}^{-1}\mathbf{r}_{k-2})$
$\mathbf{C}\mathbf{p}_k = \mathbf{C}^{-1}\mathbf{r}_{k-1} + \beta_k \mathbf{C}\mathbf{p}_{k-1}$
end
(c) $\alpha_k = (\mathbf{C}^{-1}\mathbf{r}_{k-1})^T(\mathbf{C}^{-1}\mathbf{r}_{k-1}) \,/\, (\mathbf{C}\mathbf{p}_k)^T(\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1})(\mathbf{C}\mathbf{p}_k)$
(d) $\mathbf{C}\mathbf{x}_k = \mathbf{C}\mathbf{x}_{k-1} + \alpha_k \mathbf{C}\mathbf{p}_k$
(e) $\mathbf{C}^{-1}\mathbf{r}_k = \mathbf{C}^{-1}\mathbf{r}_{k-1} - \alpha_k(\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1})\mathbf{C}\mathbf{p}_k$
end
Step 2. $\mathbf{C}\mathbf{x} = \mathbf{C}\mathbf{x}_k$

Defining preconditioner $\mathbf{M}$ as $\mathbf{M} = \mathbf{C}^2$ (also positive definite) and allowing $\mathbf{z}_k$ to be the solution of the system $\mathbf{M}\mathbf{z}_k = \mathbf{r}_k$, the above algorithm simplifies to Algorithm 3.1. Note that $\mathbf{C}\mathbf{z}_k = \mathbf{C}^{-1}\mathbf{r}_k$.

Algorithm 3.1 [Preconditioned Conjugate Gradients] Given a symmetric positive definite, $\mathbf{A} \in R^{n\times n}$ and $\mathbf{b} \in R^n$, the following algorithm solves the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ using the method of conjugate gradient with preconditioner $\mathbf{M} = \mathbf{C}^2 \in R^{n\times n}$

Step 0. Set (a) k = 0; (b) $\mathbf{x}_0 = \mathbf{0}$; (c) $\mathbf{r}_0 = \mathbf{b}$
Step 1.**While** $\mathbf{r}_k \neq \mathbf{0}$
(a) Solve $\mathbf{M}\mathbf{z}_k = \mathbf{r}_k$
(b) k = k + 1
(c) **if** k = 1
$\mathbf{p}_1 = \mathbf{z}_0$
else
$\beta_k = \mathbf{r}_{k-1}^T\mathbf{z}_{k-1} \,/\, \mathbf{r}_{k-2}^T\mathbf{z}_{k-2}$
$\mathbf{p}_k = \mathbf{z}_{k-1} + \beta_k \mathbf{p}_{k-1}$
end
(d) $\alpha_k = \mathbf{r}_{k-1}^T\mathbf{z}_{k-1} \,/\, \mathbf{p}_k^T\mathbf{A}\mathbf{p}_k$
(e) $\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k$
(f) $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{A}\mathbf{p}_k$
end, Step 2.$\mathbf{x} = \mathbf{x}_k$

where $\mathbf{z_k}$ is the solution of the system $\mathbf{Mz_k} = \mathbf{r_k}$.

A number of important observations should be made about this procedure:

- It can be shown that the residuals and search directions satisfy

$$\mathbf{r}_j^T \mathbf{M}^{-1} \mathbf{r}_i = 0 \qquad i \neq j$$
$$\mathbf{p}_j^T (\mathbf{C}^{-1} \mathbf{A} \mathbf{C}^{-1}) \mathbf{p}_i = 0 \quad i \neq j$$

- The denominators $\mathbf{r}_{k-2}^T \mathbf{z}_{k-2} = \mathbf{z}_{k-2}^T \mathbf{M} \mathbf{z}_{k-2}$ never vanish because $\mathbf{M}$ is positive definite.
- Although the transformed $\mathbf{C}$ figured heavily in the derivation of the algorithm, its action is only felt through the preconditioner $\mathbf{M} = \mathbf{C}^2$. The latter equation gives the indication to keep $\mathbf{C}^2$ as "simple" as possible.
- For PCCGM to be an effective, sparse matrix technique, linear systems of the form $\mathbf{Mz} = \mathbf{r}$ must be easily solved and convergence must be rapid.

3.6 Variants of the Conjugate Gradient Method

Several implementations of the CGM are available as part of mathematical software packages. Some of these implementations, such as LSQR, CGLS, LSCG, LSLQ, RRLS, GRAIG, and RRLSQR, are described in Paige and Saunders, (1982). In the initial description of the CGM the expression used for the computation of the step-size parameter β_k was

$$\beta_k = \frac{\mathbf{r}_k^T \mathbf{p}_k}{\mathbf{p}_{k-1}^T \mathbf{p}_{k-1}}.$$

One of the well known variants of CGM is that proposed by Fletcher-Reeves in which β_k is computed as

$$\beta_k = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}. \qquad \textbf{(Fletcher-Reeves)}$$

The previous descriptions of the conjugate gradient algorithm are based on that formula. The implementation of the CGM used to obtain the results reported in this chapter uses the same formula even though, in our experience, only modest performance differences were observed between the three variants.

Another important method is the Polak-Ribiera method, where

$$\beta_k = \frac{(r_k^T - r_{k-1})^T r_k}{r_{k-1}^T r_{k-1}} \qquad \textbf{(Polak-Ribiera)}$$

is used to determine β_k.

Other variants of the CGM are the partial and the asymmetric version . In the partial CGM, the conjugate gradient procedure is carried out for $m + 1 < n$ steps (n is the number of variables in the system of equations considered) and then, rather than continuing, the process is restarted from the current point and $m + 1$, conjugate gradient steps are taken. The asymmetric version is slightly more efficient than the symmetric one since it avoids m square root operations.

In general, the conjugate gradient method is an efficient and fast solver for large sparse systems of equations, if the condition number of the system solved is small. The algorithm, for example, converges extremely quickly if $\mathbf{A}$ is of the form $\mathbf{M} - \mathbf{N}$, where $\mathbf{M}^T\mathbf{M} = \mathbf{I}$ and $\mathbf{N}$ has low rank. If the system is ill-conditioned the number of iterations in CGM will be too large and often convergence will not occur. In CGM, the CPU time required is in the order of CGM iterations multiplied by $O(n^2)$. In order to overcome the ill-conditioning and to reduce the number of CGM iterations and CPU time, the preconditioned conjugate gradient method (PCCGM) is used, [Ponnambalam, et al., (1992)].

The choice of a good preconditioner can have a dramatic effect upon the rate of convergence. Some of the possibilities will be discussed in the next section.

3.7 Preconditioning

In this section, the use of preconditioning is examined in terms of the reduction of the number of iteration steps required to obtain a good approximation to the solution **x** of equation $\mathbf{Ax} = \mathbf{b}$. The best preconditioner is given by the inverse of **A**. Then the solution is attained in one iteration step. The amount of work to construct the inverse, however, could be excessively high in practical circumstances. Consequently, preconditioning must be regarded as the optimum between the cost of constructing and manipulating the preconditioner and the acceleration of the iteration process. The three preconditioning methods used in the tests of this chapter have proven valuable in the last decade. The first method generates matrices which, when used as preconditioners before the conjugate gradient algorithm is called to compute projections, keep the positive definite property of matrix **A**. On the other hand, the last two methods are used in the actual process of the conjugate gradient algorithm.

3.7.1 Scaling by the Diagonal of A

The simplest form of preconditioning is the scaling of the rows and columns of matrix **A**, with e.g. the intention of obtaining a unit diagonal , rows, or columns of equal norm, or, in special cases, a symmetric system. The scaling by the diagonal of **A** is in some respects optimal, since it approximately minimizes the condition number of $\mathbf{D}^{-1}\mathbf{A}$ among all diagonal scalings, [Van der Sluis, (1969)]. Scaling by the diagonal of **A** also has the advantage of reducing the number of multiplications within an iteration step, [Meyerink & Van der Vorst, (1977)].

3.7.2 Incomplete LL^T (Cholesky) Factorization for Positive Definite Matrices

A method for the solution of $\mathbf{Ax} = \mathbf{b}$ is obtained using the Cholesky decomposition

$$\mathbf{A} = \mathbf{LDL}^T$$

where $\mathbf{L}$ is a lower triangular matrix with unit diagonal elements. If $\mathbf{A}$ is a large sparse matrix, this decomposition has a drawback, because, in general, $\mathbf{L}$ does not reflect the sparsity of $\mathbf{A}$, unless the minimum degree ordering procedure is used. The use of the minimum degree ordering procedure is proved to reduce not only "fill-in" but also the CPU time and the number of floating point operations required for computations with sparse matrices, [Lindfield & Penny, (1995)]. It is observed, however, that the entries of $\mathbf{L}$ corresponding to the zero values of $\mathbf{A}$ are usually small. Hence, putting these entries equal to zero may yield a reasonable approximation of $\mathbf{A}$,

$$\mathbf{A} + \mathbf{R} = \mathbf{L'DL'}^T,$$

where $\|\mathbf{R}\|$ is hopefully small compared to $\|\mathbf{A}\|$ and $\mathbf{L'}$ is a matrix with near zero entries set to zero.

In the incomplete factorization methods, elements of $\mathbf{L}$ are replaced by zero during the factorization process (see Appendix B for computation details of $\mathbf{L}$). There is a choice, then, to allow for some fill-in. For example, if $\mathbf{A}$ is a sparse structured band matrix originating from the discretized Poisson equation, one could allow for the fill-in of s extra diagonals in $\mathbf{L}$, leading to the IC(s) method, [Meyerink & Van der Vorst, (1977); Meyerink & Van der Vorst, (1981)]. Of course, IC(s) with $s > 0$ will yield a better approximation of A than IC(0) at the cost of increased storage and computation time. Meyerink and Van der Vorst proved these factorization processes do not break down in quite general circumstances. The main advantage of using incomplete Cholesky factors for preconditioning is that these factors preserve the sparsity structure of the original constraint matrix. This is very crucial when the constraint matrix is large but relatively sparse. Such matrices arise often in large-

scale linear programming and, particularly, in implementations of interior point algorithms for large multicommodity network flow problems, [Setiono, (1990)]. In the following sections, the experimental results from the implementation of two different variants of incomplete Cholesky factors, [Golub & Van Loan (1983); Setiono, (1990)], will be reported.

The difference between the two variants lies only in the way the incomplete Cholesky factor is defined. In the Van Loan variant the diagonal elements of the factorization matrix are simply the square root of the diagonal elements of **A**. Setiono defines the same elements by the formula

$$L_{(ii)} = \sqrt{L_{(ii)} - \sum_{k=1}^{i-1} L_{(ik)}^2}, \qquad i = 1, 2, ..., n.$$

The above definition affects all the entries of the incomplete Cholesky factor because, in the computation process, the off-diagonal elements are obtained as a function of the diagonal entries of the matrix.

3.7.3 Other Preconditioners

When one is willing to spend more computational effort in the construction of the preconditioner, other methods can also be considered. For example, incomplete block factorization, [Axelsson & Lindskog, (1986); Concus, et al., (1985)], line relaxation methods, and alternating direction methods, [Varga, (1962)]. In these methods, a (simple) implicit system must be solved in each iteration step.

The last preconditioner for the CG method mentioned in this section is that of scaling the rows and columns of matrix **A** by a diagonal matrix with entries formed from the two-norm of the rows of **A**, [Kortanek, (1993)]. The preconditioner was implemented by Kortanek in FORTRAN code and run under the vector super computer CRAY, solving LP problems by the primal affine scaling algorithm. This preconditioning method combined with

a conjugate gradient-based implementation was reported to give good results for ill-conditioned problems which had up to 7700 equations and as many as 9000 variables. However, in the limited experiments performed for this study Kortanek preconditioner did not seem to pay for itself compared to the diagonal preconditioner.

3.8 Computational Experience

This section will report some numerical results on the performance of the preconditioned conjugate gradient method on a set of randomly generated problems. Both simple (diagonal) and more complicated preconditioners (incomplete Cholesky) were implemented and tested in the experiments performed for this chapter, so one has a clear comparison of the performance of the PCCGM regarding CPU time, floating point operations, number of iterations, and solution accuracy. The tests were carried out in MATLAB ver. 4.2 on a networked SUN SPARC workstation.

For purposes of comparison, the number of floating point operations recorded was obtained by the function FLOPS of MATLAB. A disadvantage exists in the use of FLOPS as a performance criterion; this function, which totals the number of floating point operations performed, does not distinguish between the type of operations performed. Therefore, an algorithm that has more additions and subtractions than multiplications and divisions will perform quicker than a method where the latter operations are prevalent.

To record the execution time for each preconditioner, the time functions ETIME, CPUTIME, and TIC-TOC were used. These functions, as with all computer timing functions, are not accurate. The results obtained can differ from one run of the same problem to the next and may not settle until a third or fourth run of the problem has been executed. Another thing affecting the accuracy of the timing results is the networked computer. As all the methods were tested on a networked computer, processing speed varied and was dependent on the workload on the network at those given times. For these reasons, the number of floating point operations have been provided instead of the timing results.

3.8.1 Dense and Sparse Matrices

When a matrix has only a few non-zero entries, then it is convenient to declare that matrix as a sparse one. Declaring and treating a matrix as sparse results in the matrix not being stored as a whole, but being built from the non-zero elements and their indexes every time it is used. Additionally, no operations with the zero entries of the matrix are then performed. MATLAB has such an option as a built-in function (SPARSE) which converts full matrices to sparse or generates sparse matrices using various arguments. To observe the effect of sparsity on the conjugate gradient method and on different preconditioners experiments were performed with both the SPARSE option on and off for three levels of density and five condition numbers for the **A** matrix. The chosen densities were 12%, 20%, and 50%. In the following table, some of the results for density equal to 20% and condition number equal to 100 are presented.

Size No Variables	Storage Dense/Sparse	Setiono Iter/Flops	Van Loan Iter/Flops	Diagonal Iter/Flops	No Iter/Flops
10	Sparse	1/0.62E3	5/1.78E3	9/3.19E3	10/3.30E3
10	Dense	1/3.79E3	3/5.45E3	10/16.5E3	10/6.12E3
30	Sparse	9/3.12E4	12/2.93E4	30/4.32E4	29/4.05E4
30	Dense	46/89E4	12/23E4	30/39E4	29/12E4
80	Sparse	37/1.60E6	28/1.03E6	52/0.35E6	40/0.26E6
80	Dense	128/30.8E6	24/6.1E6	49/6.4E6	43/1.1E6
100	Sparse	59/5.05E6	29/2.23E6	65/0.66E6	42/0.42E6
100	Dense	173/76.4E6	27/12.7E6	61/1.2E6	44/1.8E6

Table 3.1 Comparative Results For Sparse and Dense Problems

As one can conclude studying the results recorded in Table 3.1, working with sparse matrices gives savings from 85% to 328% when no preconditioner is used. The savings for the Setiono variant, the diagonal, and the Van Loan variant preconditioners are from 511% to 1400%, 417% to 1728%, and 206% to 684%, respectively. Savings on floating point

operations performed depend, as expected, on the level of sparsity and the size of the matrix. The larger and more sparse the matrix is, the greater the savings. One can expect the same savings on the CPU-time required by each preconditioning technique. The timing results achieved, even though not so valuable in terms of accuracy for the reasons mentioned earlier, confirm this. All sparse matrices used in the experiments were generated by the SPRANDSYM function of MATLAB which generates sparse, symmetric, positive definite matrices. The density and condition number of these matrices can be specified by the user. To make sure a fair comparison between the two storage strategies was established, the same matrices were used for the tests with the sparse option off. The full-storage matrices are obtained from the sparse ones by converting them using the FULL function of MATLAB.

3.8.2 Preconditioners and Sparsity

To study the effect of sparsity on the chosen preconditioning techniques, it was necessary to test them on matrices with different levels of density. All matrices are sparse and are generated as in the previous section. Tables 3.2 and 3.3 pertain to matrices with condition number equal to 100 and size 50 and 100, respectively.

Density	Setiono Iter/Flops	Van Loan Iter/Flops	Diagonal Iter/Flops	No Iter/Flops
12%	6/4.17E4	20/8.75E4	33/7.82E4	35/8.06E4
20%	53/6.24E5	19/1.84E5	35/1.12E5	34/1.05E5
50%	139/5.35E6	37/2.38E5	37/2.38E5	34/2.11E5

Table 3.2 Comparative Results for a Matrix of Size 50

Density	Setiono Iter/Flops	Van Loan Iter/Flops	Diagonal Iter/Flops	No Iter/Flops
12%	38/1.55E6	25/8.73E5	56/3.96E5	42/2.91E5
20%	63/46.87E5	33/21.56E5	57/5.80E5	40/3.98E5
50%	143/37.89E6	35/8.70E6	61/1.33E6	43/0.91E6

Table 3.3 Comparative Results for a Matrix of Size 100

From the experience gathered and from studying the results in the above two tables, the following observations are apparent

- For very sparse systems (12%), the technique based on the Setiono variant of the incomplete Cholesky preconditioner stands out as the clear winner. The results are very accurate and the error is less than 10^{-9}.
- For less sparse matrices, the Van Loan variant preconditioning method outperforms the Setiono, again providing very accurate results.
- The preconditioner technique based on the diagonal of the matrix is cheaper, but along with the standard conjugate gradient method, gives less accurate results. The error pertaining to the last two columns in the tables is 10^{-4}.

3.8.3 Preconditioners and Condition Numbers

Ill-conditioned problems are hard to solve. Preconditioning techniques alter the condition number of a matrix in order to first solve the problem and, then, to reduce the number of iterations required by the algorithm. Tests were performed to study the behaviour of the preconditioners on sparse matrices with different condition numbers. Table 3.4 pertains to matrix of size 30 and density 12%.

Condition No	Setiono Iter/Flops	Van Loan Iter/Flops	Diagonal Iter/Flops	No Iter/Flops
10E2	4/0.1E5	11/0.16E5	25/0.29E5	29/0.33E5
10E5	3/0.08E5	12/0.18E5	72/0.84E5	50/0.58E5
10E8	3/0.07E5	16/0.21E5	**	**
10E11	19/0.38E5	21/0.30E5	**	**
10E14	31/0.64E5	20/0.29E5	**	**

Table 3.4 Comparative Results for Matrices of Different Condition Number

As can be seen in Table 3.4, the standard (unpreconditioned) conjugate gradient method, along with the diagonal preconditioner, fails to obtain an accurate solution when the matrices have a condition number greater than 10^5. Even for smaller condition numbers, more sophisticated preconditioners (variants of the incomplete Cholesky factorization) need less iterations, less floating point operations, and, according to the test results, give more accurate solutions. In tests with matrices of bigger sizes (50, 80, 100) the Van Loan variant technique was found to outperform the Setiono version. The latter needs more than 1000 iterations to converge for matrices with condition number 10^5 or greater.

3.9 Conclusions

In this chapter three different preconditioners were considered which were used in the implementation of the conjugate gradient method. The variants were tested, on randomly generated systems of linear equations, taking sparsity into account.

From the numerical results presented in this chapter and the experience garnered, some interesting aspects of the preconditioning techniques tested can be highlighted.

- Both the Setiono and the Van Loan techniques give very accurate results but they are expensive in CPU time and floating point operations.

- For ill-conditioned problems with a condition number greater than 10^5, only the Setiono and the Van Loan techniques converge to accurate solutions.
- When sparse matrices are considered with density less than 15%, the Setiono variant is very attractive. The factorization pays for itself and the number of flops is usually less than any other variant.
- If the matrices are ill-conditioned and denser, the Van Loan technique performs better than the Setiono variant.
- The diagonal preconditioner demands the least CPU time but, appears to perform very poorly as far as accuracy is concerned.
- As the results of the tests show, preconditioners can, at times, reduce the iteration count significantly and reduce the execution time less frequently.
- The use of preconditioners is justified only in sparse problems or in systems where the standard conjugate gradient method fails due to ill-conditioning.

Chapter 4

Interior Point Methods for Linear Programming

4.1 Introduction

After ten years of development in both interior point and simplex methods, the mathematical community is now at a point at which both approaches can handle problems that seemed intractable before. Interior point methods seem to be superior to simplex algorithms for many large-scale, sparse, linear programming problems, [Todd, (1994)]. For a good survey of the significant developments in the field of interior point methods for linear programming the reader is referred to Lustig, et al., (1994).

The present chapter is concerned with studying recent interior point methods which introduce considerable new developments in the field of linear programming. The performance of this selection of algorithms on a range of sparse and ill-conditioned problems is also analysed.

The described algorithms are based on interior point methods but new ideas are introduced to improve their performance. The new mathematical technique of the reciprocals, [Karmarkar & Ramakrishnan, (1991)], is combined with the basic idea of the projective algorithm of Karmarkar, [Karmarkar, (1984)]. The primal-dual logarithmic barrier method, [Kojima, et al., (1989); Megiddo, (1989)], was developed into the predictor corrector algorithm, [Mehrotra, (1992)]. Finally, the affine scaling algorithm, [Adler, et al., (1989); Barnes, (1986); Vanderbei, et al., (1986)], was modified to solve problems without interior points, [Andersen, (1993)].

4.2 A Dual Variant of the Karmarkar Algorithm

The first of the algorithms described in this chapter is a variant of the projective algorithm, [Karmarkar, (1984)], applied to the dual of the LP. Asymptotically, the variant becomes identical to the Karmarkar dual projective algorithm, [Karmarkar & Ramakrishnan, (1991)]. The direction finding step of the variant of the dual projective algorithm consists of two major steps.

Step 1. Perform an affine scaling transformation and take the projected steepest ascent step.

Step 2. Apply the reciprocal estimates to obtain a primal estimate for the current dual iterate, and take a step in the dual space to improve the primal estimate.

The effect of Step 2 is to move the iterate closer to the central trajectory. The central trajectory is defined as a locus of points where the gradient of the objective function and the gradient of the logarithmic potential function, $F(\mathbf{s}) = \sum_{i=1}^{n} \ln(s_i)$, where s_i are slack variables in the dual, are parallel to each other, [Bayer, (1989)]. The theory of the reciprocal estimates states that, on the central trajectory, the reciprocals of the slack variables are, up to a scaling constant, feasible primal estimates. Thus, improving the primal estimates has the effect of moving the iterates closer to the central trajectory. Barnes, et al., (1988) adopted a strategy of repeated centring combined with affine scaling, and proved the convergence of such a method.

The algorithm consists of two phases. In Phase 1, the dual phase, the dual of the LP problem is solved. In Phase 2, a primal solution is obtained, starting with the estimate from the converged dual solution. At the end of the execution of both phases, a primal-dual pair with a small relative duality gap is obtained.

The reciprocal estimate method used in the algorithm is a new mathematical technique for estimating the primal variables, [Adler, (1989)]. Consider the LP problem is given in standard equality form without upper bounds. Let $(\mathbf{y}^*,\mathbf{s}^*)$ be a point on the central trajectory in the dual polytope. The level set of the dual objective function containing $\mathbf{y}^*$ is

defined by $\mathbf{b}^T\mathbf{y} = \mathbf{b}^T\mathbf{y}^*$. On this central trajectory, the optimality condition for the potential function states that $\mathbf{A}\mathbf{g}^*$ is parallel to $\mathbf{b}$; i.e.,

$$\mathbf{A}\mathbf{g}^* = k\mathbf{b}$$

where,

$$g_i^* = \frac{1}{s_i^*}.$$

This implies that the reciprocals of slacks are, up to a scale constant factor k, feasible solutions to the primal problem.

A similar derivation to the above applies to the primal problem with upper bounds. Although one is not necessarily on the central trajectory at the termination of the dual algorithm, the vector of reciprocals of slacks can still be used as a good starting point for the primal feasibility algorithm, since one can scale this vector by a scalar, σ, to account for not being exactly on the central trajectory. Consider now that there is a strictly interior dual feasible solution at hand. This is true after the dual phase of Algorithm A terminates. At this point the reciprocal estimate algorithm, Algorithm B, can be called to estimate the primal variables. A detailed description of both Algorithms A and B can be found in Appendix C.

Algorithm B consists of two phases. In the first phase, an initial primal solution, $\mathbf{x}^0$, is evaluated using the reciprocal estimates. In the second phase, this solution is made feasible using an iterative refinement algorithm. The initial estimate of vector $\mathbf{x}^0$ is composed as a linear combination of two vectors $\bar{\mathbf{x}}$ and $\sigma\mathbf{x}'$, where $\mathbf{x}'$ is the vector of reciprocal of slacks. Vector $\bar{\mathbf{x}}$ is composed from the lower and upper bounds of the problem depending on the values of the slack variables of the dual problem. The linear combination $\bar{\mathbf{x}} + \sigma\mathbf{x}'$, where σ is chosen so as to make $\mathbf{x}^0$ as close to $\mathbf{b}$ as possible, determines whether the primal estimate for variable x_i should be σ/s_{1i} or $\mathbf{u}-\sigma/s_{2i}$, where $\mathbf{u}$ is an upper bound of the solution vector. This decision is based on which of the reciprocal is smaller. To ensure that $\mathbf{x}^0$ satisfies the bounds $\mathbf{0} < \mathbf{x} < \mathbf{u}$ an upper bound of σ is given, which is then appropriately adjusted.

After Algorithm B terminates, it provides a pair of dual feasible and primal feasible solutions. For this pair of solutions, the relative duality gap is smaller than a prescribed small number ε.

A key feature of the algorithm is that a preconditioned conjugate gradient method is used although Karmarkar and Ramakrishnan do not provide any precise form of the preconditioner used in their experiments in their paper.

4.3 An Affine Scaling Algorithm and the Big-M Method

As mentioned in Chapter 1, the affine scaling algorithm was originally proposed by Dikin as early as in 1967 and later rediscovered by Vanderbei, et al., (1986).

Barnes, (1986) considered the LP problem in its standard form and its dual. Given a feasible point $\mathbf{y}$ to the primal and a scalar $0 < \omega < 1$, Barnes showed that the ellipsoid

$$\sum_{i=1}^{n} (x_i - y_i)^2/y_i^2 \leq \omega$$

lies in the positive orthant. Solving the problem

$$\begin{aligned} &\textbf{Min} \quad \mathbf{c}^T\mathbf{x} \\ &\textbf{s.t.} \quad \mathbf{Ax} = \mathbf{b} \\ &\sum_{i=1}^{n}(x_i - y_i)^2/y_i^2 \leq \omega \\ &\mathbf{x} \geq \mathbf{0} \end{aligned}$$

leads to a point $\mathbf{x}$ such that $\mathbf{c}^T\mathbf{x} < \mathbf{c}^T\mathbf{y}$. An iterative process is then constructed as follows. If $\mathbf{x}^{(0)} > \mathbf{0}$ and $\mathbf{A}\mathbf{x}^{(0)} = \mathbf{b}$, then after iteration k where $\mathbf{x}^{(k)}$ is calculated, set $\mathbf{D} = \text{diag}(x_j^{(k)})$, j = 1, ..., n and find $\mathbf{x}^{(k+1)} > \mathbf{0}$ from the relation

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \omega \frac{\mathbf{D}_k^2(\mathbf{c} - \mathbf{A}^T\lambda_k)}{\|\mathbf{D}_k(\mathbf{c} - \mathbf{A}^T\lambda_k)\|}$$

where $\lambda_k = (\mathbf{A}\mathbf{D}_k^2\mathbf{A}^T)^{-1}\mathbf{A}\mathbf{D}_k^2\mathbf{c}$ is a dual feasible solution.

As described above, the algorithm must start with an interior feasible point. In order to satisfy that requirement one must add a new column $\mathbf{a}_{n+1}$ to the constraint matrix $\mathbf{A}$ and an extra corresponding variable x_{n+1} to $\mathbf{x}$. If it is defined $\mathbf{a}_{n+1} = \mathbf{b} - \sum_{i=1}^{n}\mathbf{a}_i$, the constraints

$\mathbf{Ax} = \sum_{i=1}^{n+1} \mathbf{a_i x_i} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ are satisfied by the vector $\mathbf{x} = (1, 1, ..., 1)^T \in E^{n+1}$. If one then adds a large positive component c_{n+1} corresponding to x_{n+1} to $\mathbf{c}$, the optimum solution to the constructed problem will have $x_{n+1} = 0$ and the variables $(x_1, x_2, ..., x_n)$ will then form the solution to the original problem. To ensure convergence to the optimal solution of the original problem, c_{n+1} must be equal to a sufficiently large number M so that $x^{n+1} \rightarrow 0$, hence the name big-M.

4.4 An Infeasible Dual Affine Scaling Approach

Another algorithm based on affine scaling will be described in this section. The method can be viewed as dynamic a "big-M" method but it actually never calculates the "big-M", [Andersen, (1993)]. The main advantage of the algorithm is that in contrast to the "big-M" method, this approach always finds a feasible point, even for problems without interior points.

The proposed dual for a LP in standard form is

$$\begin{aligned} &\textbf{Max } (\mathbf{b}^T\mathbf{y} - My_{ar}) \\ &\textbf{s.t. } \begin{bmatrix} \mathbf{A}^T & -\mathbf{e} \\ \mathbf{0} & -1 \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ y_{ar} \end{bmatrix} + \begin{bmatrix} \mathbf{s} \\ s_{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{c} \\ 0 \end{bmatrix} \\ &\quad \mathbf{s} \geq \mathbf{0},\ s_{n+1} \geq 0 \end{aligned} \qquad (4.4.1)$$

where y_{ar} measures the infeasibility of the current dual point, hence the name infeasible dual method. If $y_{ar} = 0$ then $\mathbf{y}$ is feasible and the algorithm is equivalent to the big-M method. The corresponding primal problem to problem (4.4.1) is

$$\begin{aligned} &\textbf{Min } \mathbf{c}^T\mathbf{x} \\ &\textbf{s.t. } \begin{bmatrix} \mathbf{A} & 0 \\ -\mathbf{e}^T & -1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ x_{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ -M \end{bmatrix} \\ &\quad \mathbf{x} \geq \mathbf{0},\ x_{n+1} \geq 0 \end{aligned} \qquad (4.4.2)$$

For this problem $-\mathbf{e}^T\mathbf{x} - x_{n+1} = -M$ or, since $x_{n+1} \geq 0$, $M > \mathbf{e}^T\mathbf{x}$. The algorithm for the above pair of primal-dual formulation is as follows:

Step 0. Set k = 0
Step 1. **While** optimality_criterion = False

(a) $\mathbf{s^k = c - A^T y^k + y^k_{ar} e;}$

(b) $\Delta y_{ar} = -C(y^k_{ar})^2;$

(c) $\Delta \mathbf{y} = (\mathbf{AS_k^{-2}A^T})^{-1}(\mathbf{b} + \Delta y_{ar}\mathbf{AS_k^{-2}e});$

(d) $\mathbf{x^k = S_k^{-2}(A\Delta y - \Delta y_{ar} e);}$

(e) $\Delta \mathbf{s} = -(\mathbf{A^T \Delta y} - \Delta y_{ar}\mathbf{e});$

(f) **if** ($\Delta \mathbf{s} \geq \mathbf{0}$ and $\Delta \mathbf{s} \neq \mathbf{0}$) return unbounded; **end**

(g) $\alpha = \min\{-s_i^k/\Delta s_i \mid \Delta s_i < 0, i = 1,..., n\};$

(h) $\mathbf{y^{k+1} = y^k + \lambda \alpha \Delta y;}$

(i) $y_{ar}^{k+1} = y_{ar}^k + \lambda\alpha\Delta y_{ar};$

(j) **if** $y_{ar}^{k+1} \leq 0$ set $y_{ar}^{k+1} = 0$

(k) k = k + 1;

end

where $\mathbf{S_k} = \text{diag}(s_1, s_2, ..., s_n)$ and $0 < \lambda < 1$. The key property of the algorithm is that $x^k_{n+1} = C > 0$ is kept fixed, instead of M. In each iteration M is set dynamically in such a way that $M > \mathbf{e^T x}$ is satisfied.

The direction for this algorithm is given by the equations

$$\Delta \mathbf{y} = (\mathbf{AS_k^{-2}A^T})^{-1}\mathbf{b} + \Delta y_{ar}(\mathbf{AS_k^{-2}A^T})^{-1}\mathbf{AS_k^{-2}e}$$

$$\Delta y_{ar} = (-M + \mathbf{e^T S_k^{-2} A^T \Delta y})/(s^{-2}_{n+1} + \mathbf{e^T S_k^{-2} e})$$

and is a linear combination of the feasibility direction $(\mathbf{AS_k^{-2}A^T})^{-1}\mathbf{AS_k^{-2}e}$ and the steepest ascent direction for the problem without the artificial variable y_{ar}. The algorithm avoids possible numeric instability that may be caused by a large M by setting Δy_{ar} before the calculation of $\Delta \mathbf{y}$.

4.5 The Predictor Corrector Method

The last of the interior point methods described in the present chapter can be viewed as a relaxation of the well-known primal-dual method, [Lustig, (1991)]. A minor variant of

the algorithm as described in this section is the current algorithm implemented in OB1, one of the most sophisticated implementations of interior point methods, [Todd, (1994)].

Consider a linear program in standard form and its dual. The predictor corrector method can be derived directly by applying the logarithmic barrier method to the dual of the problem

$$\begin{aligned} &\textbf{Max} \quad \mathbf{b}^T\mathbf{y} + \mu \sum_{j=1}^{n} \ln z_j \\ &\text{s.t.} \quad \mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{c}, \end{aligned}$$

where $\mathbf{z}$ are slack variables and $\mu > 0$.

The Lagrangian of this problem is

$$\mathbf{L}(\mathbf{x}, \mathbf{y}, \mathbf{z}, \mu) = \mathbf{b}^T\mathbf{y} + \mu \sum_{j=1}^{n} \ln z_j - \mathbf{x}^T(\mathbf{A}^T\mathbf{y} + \mathbf{z} - \mathbf{c}). \qquad (4.5.1)$$

The first order optimality conditions for (4.5.1) are

$$\begin{aligned} &\mathbf{XZe} = \mu\mathbf{e} \\ &\mathbf{Ax} = \mathbf{b} \\ &\mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{c} \end{aligned} \qquad (4.5.2)$$

where $\mathbf{X}$ and $\mathbf{Z}$ are diagonal matrices whose diagonal elements are the variables x_j and z_j respectively. By substituting $\mathbf{x} + \Delta\mathbf{x}$, $\mathbf{y} + \Delta\mathbf{y}$, and $\mathbf{z} + \Delta\mathbf{z}$ in (4.5.2), it is desirable that the new estimates satisfy

$$\begin{aligned} &(\mathbf{X} + \Delta\mathbf{X})(\mathbf{Z} + \Delta\mathbf{Z})\mathbf{e} = \mu\mathbf{e}, \\ &\mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b}, \\ &\mathbf{A}^T(\mathbf{y} + \Delta\mathbf{y}) + \mathbf{z} + \Delta\mathbf{z} = \mathbf{c}. \end{aligned} \qquad (4.5.3)$$

Collecting terms gives the system

$$\mathbf{X}\Delta\mathbf{z} + \mathbf{Z}\Delta\mathbf{x} = \mu\mathbf{e} - \mathbf{XZe} - \Delta\mathbf{X}\Delta\mathbf{Ze}, \qquad (4.5.4a)$$

$$\mathbf{A}\Delta\mathbf{x} = \mathbf{b} - \mathbf{Ax}, \qquad (4.5.4b)$$

$$\mathbf{A}^T\Delta\mathbf{y} + \Delta\mathbf{z} = \mathbf{c} - \mathbf{A}^T\mathbf{y} - \mathbf{z}. \qquad (4.5.4c)$$

One can notice that (4.5.4) is similar to (4.5.2) with the exception of the non-linear term $\Delta\mathbf{X}\Delta\mathbf{Ze}$ in (4.5.4). Mehrotra, (1992) proposed first solving, for current values of $\mathbf{x}$ and $\mathbf{z}$, the affine system (4.5.5), which is linear for $\Delta\mathbf{x}'$, $\Delta\mathbf{y}'$, $\Delta\mathbf{z}'$,

$$
\begin{aligned}
&\mathbf{X}\Delta\mathbf{z}' + \mathbf{Z}\Delta\mathbf{x}' = -\mathbf{XZe}, \\
&\mathbf{A}\Delta\mathbf{x}' = \mathbf{b} - \mathbf{Ax}, \\
&\mathbf{A}^{\mathbf{T}}\Delta\mathbf{y}' + \Delta\mathbf{z}' = \mathbf{c} - \mathbf{A}^{\mathbf{T}} - \mathbf{z},
\end{aligned}
\qquad (4.5.5)
$$

for $\Delta\mathbf{x}'$, $\Delta\mathbf{y}'$, $\Delta\mathbf{z}'$ and then substituting the vectors $\Delta\mathbf{x}'$ and $\Delta\mathbf{z}'$ found by solving (4.5.5) for the $\Delta\mathbf{X}\Delta\mathbf{Ze}$ term in the right-hand side of (4.5.4), which is then solved for $\Delta\mathbf{x}$, $\Delta\mathbf{y}$, $\Delta\mathbf{z}$. The updating of the solution is then carried out by the formulae

$$
\begin{aligned}
\mathbf{x} &= \mathbf{x} + \Delta\mathbf{x} \\
\mathbf{y} &= \mathbf{y} + \Delta\mathbf{y} \\
\mathbf{z} &= \mathbf{z} + \Delta\mathbf{z}
\end{aligned}
$$

If one continues to substitute at each step the $\Delta\mathbf{x}$ and $\Delta\mathbf{z}$ terms found by solving (4.5.4) back into the right-hand side of the (4.5.4a), the algorithm will use multiple corrections. Assuming the goal for using multiple corrections is to achieve the maximum reduction in complementarity, the optimal number of corrections for the NETLIB standard test problem afiro is four. Although multiple corrections often produce lower complementarity, each correction requires a new solution to (4.5.4). Multiple corrections have been extensively studied by Carpenter and Shanno, (1993). This study concludes that the most efficient number of corrections for a general algorithm is one.

4.6 Computational Experience

The relative performance of the described algorithms is studied in this section. Linear programming problems come in a variety of sparsity levels. Three classes of non-random test problems with different level of sparsity were chosen so that one can clearly compare their performance on fully dense (Hilbert-type), relatively sparse (Klee-Minty), and very sparse (Linear Ordering) problems.

The problems chosen are, in general, difficult to solve. Hilbert-type problems are by far the most difficult since they became highly ill-conditioned as their size grow. Linear ordering problems are related to the optimal triangulation problem which belongs in the NP-

class, [Garey & Johnson, (1979)]. Finally, the Klee-Minty problem is solved by the simplex method in exponential time.

Experiments with random generated problems were also performed. All matrices in the test problems were declared as sparse using the SPARSE function of MATLAB.

All computational experiments were performed on a SPARC networked workstation. The same functions mentioned in Chapter 3 were used for recording CPU time and floating point operations.

For all implemented algorithms the termination criterion is the absolute value of the duality gap between the primal and the dual objective function. All the runs were made with this parameter set to 5×10^{-7}.

When only the dual phase of the Karmarkar algorithm is used, the termination criterion is set to be the relative improvement in the dual objective function. This parameter was also set to 5×10^{-7}.

Even if an implementation of the predictor corrector method using multiple corrections may result, in certain types of problems, in a greater reduction of complementarity in our implementation, only one correction is used because of our intention to solve different problems without tuning each algorithm for every type of problem.

Hilbert-Type LP Problems

These are LP problems whose constraints matrix is based on the Hilbert matrix. The condition number of the Hilbert matrix is dramatically increased by its size.
The problems are of the form

$$\begin{aligned} &\textbf{min } \mathbf{c}^T\mathbf{x} \\ &\textbf{s.t. } \mathbf{A}\mathbf{x} \geq \mathbf{b}, \\ &\qquad \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

where $\mathbf{x} \in R^n$, $\mathbf{A} \in R^{n\times n}$, $\mathbf{c} \in R^n$ and $\mathbf{b} \in R^n$. Matrix $\mathbf{A}$ has entries $[a_{ij}] = [1/(i+j)]$, for i = 1, 2, ..., n and j = 1, 2, ..., n. The RHS is given by

$$b_i = \sum_{j=1}^{n} \frac{1}{i+j} \qquad i = 1, 2, ..., n.$$

The cost vector is given by

$$c_i = \frac{2}{i+1} + \sum_{j=2}^{n} \frac{1}{i+j} \qquad i = 1, 2, ..., n.$$

The primal optimum solution to these problems is $\mathbf{x}^* = (1, 1, ..., 1)^T$. Problems with n = 10, 20, 30, 40, 50 and 100 were solved and the results are recorded in Table 4.3.

The values seen on the graph of Figure 4.1 are recorded values of the objective function which are out of the range of the graph.

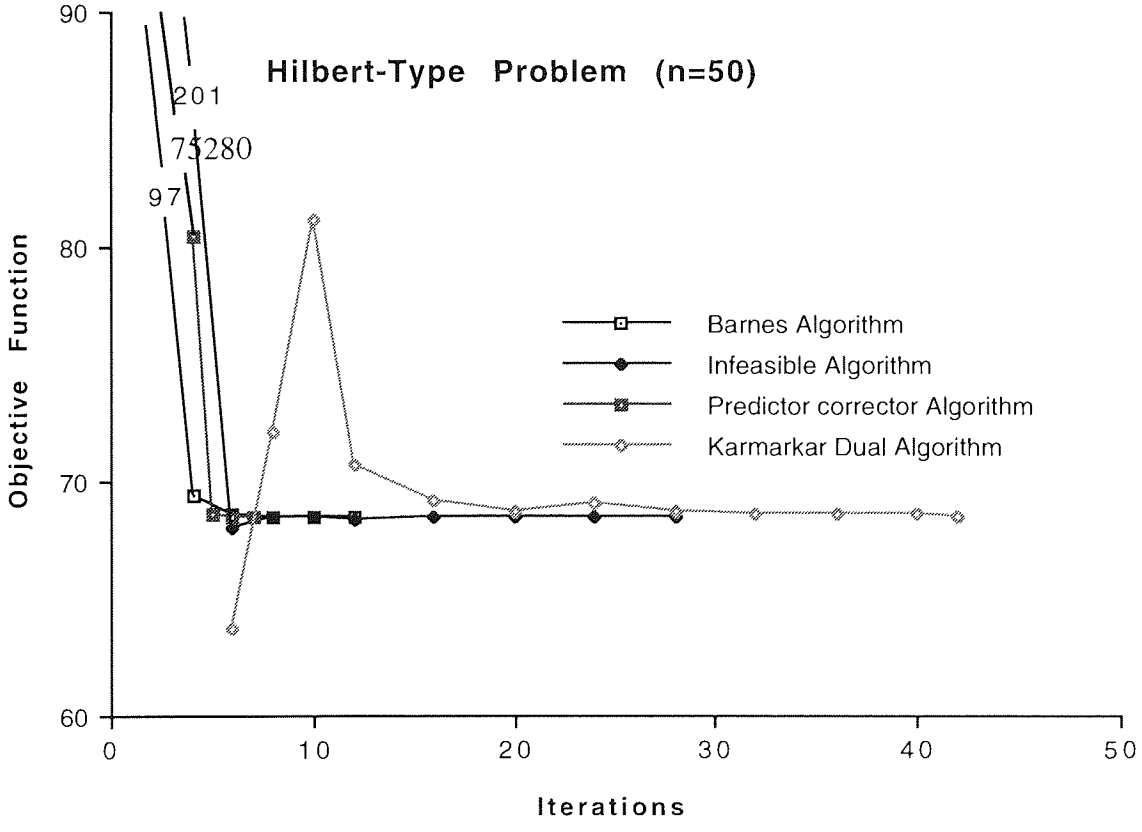

Figure 4.1 Change of the Objective Function for Hilbert Problems

Figure 4.1 is a graphical representation of the convergence of the implemented algorithms for a Hilbert-Type problem. The curve for the Karmarkar algorithm is built on the results obtained by applying the dual only phase of the described algorithm on the dual of the chosen problem. Table 4.1 records the results obtained by applying only the first phase of the algorithm to the dual of the problem and those that pertain to the solution of the original problem by applying both the dual and the primal phase.

As one can conclude from the results in Table 4.1, in all problems except for n=50, a slight improvement can be made in floating point operations by using only the first phase of the algorithm on the dual of the problem. The main advantage of this approach is that the second phase of the algorithm is avoided since it tends to become unstable in ill-conditioned problems. It must also be mentioned that the results found by solving the original problem are less accurate than these found by solving the dual.

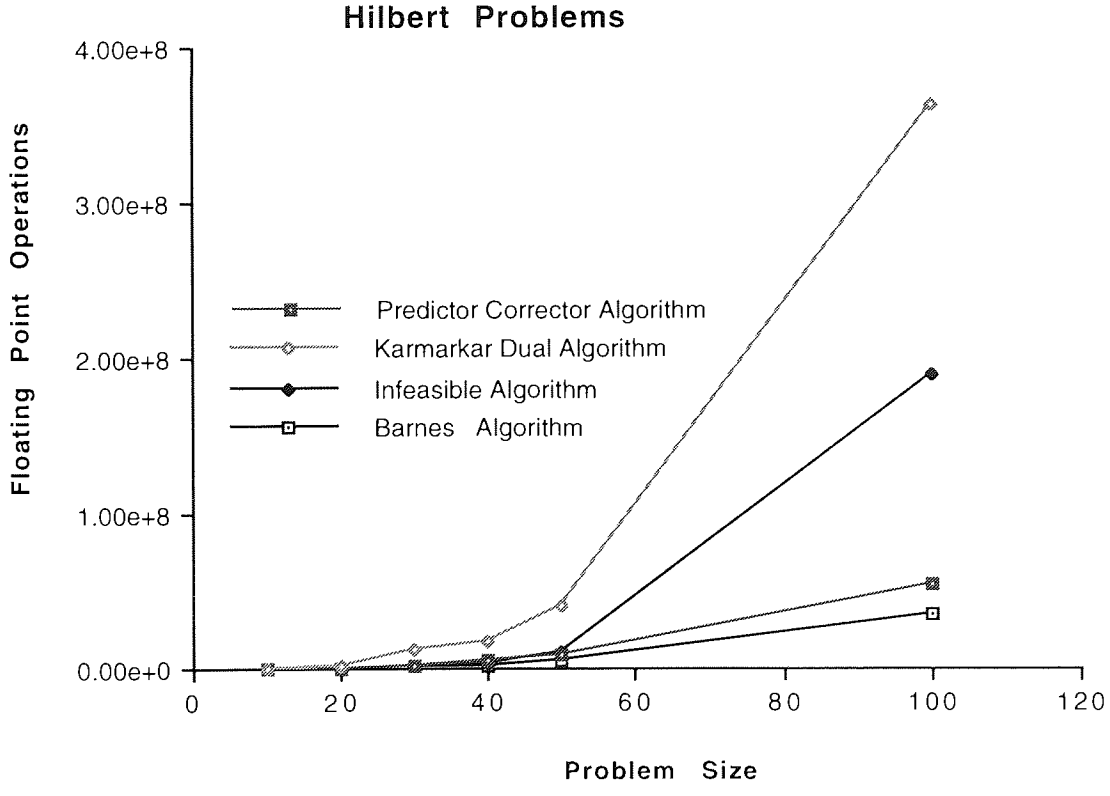

Figure 4.2 Relative Performance for Hilbert Problems

Size	Karmarkar Dual Phase			Karmarkar Primal Dual Algorithm					
	Iter	CPU(s)	Flops(10E6)	Iter_d	CPU_d	Iter_p	CPU_p	CPU(s)	Flops(10E6)
10	33	5.35	0.56	145	27.66	26	4.84	32.51	3.08
20	31	10.92	2.62	90	32.27	26	8.26	40.54	11.45
30	49	30.03	12.59	34	21.64	26	5.72	27.37	13.33
40	32	24.75	16.69	26	23.27	26	7.65	30.92	23.19
50*	43	47.66	40.87	22	23.75	26	9.93	33.68	34.78
100	51	274.71	362.84	38	244.1	26	66.66	310.84	383.42

Table 4.1 Performance of the Karmarkar Algorithm on Hilbert-Type Problems

Problem Size	Barnes Algorithm		Infeasible Algorithm		Predict Corrector Algorithm		Karmarkar Dual Variant	
Variab	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)
20	9	0.41	16	0.39	6	0.57	33	4.92
40	9	1.04	17	0.89	7	1.81	31	9.145
60	12	2.03	19	1.87	7	3.38	49	24.54
80	11	3.81	22	3.82	8	6.96	32	24.21
100	12	6.12	28	7.62	7	10.49	43	45.37

Table 4.2 Hilbert-Type Problems Results

In Table 4.2, the CPU time in seconds, required by each algorithm to solve the problem to the prescribed accuracy can by found.

From the results of that table the following observations are apparent

(i) Karmarkar's algorithm stands out as the most time consuming method, even when only the dual phase is used.

(ii) For small size problems (n = 10), the infeasible algorithm is slightly better than the Barnes algorithm, followed by the predictor corrector method.

(iii) For larger problems, the relative attractiveness of the infeasible algorithm tends to decrease rather rapidly. For n = 40 and 50, the infeasible method is outperformed by the Barnes algorithm.

(iv) For very large problems (n = 100), even the predictor corrector method performs better than the infeasible method.

From Figure 4.2, one can see that CPU times do not correspond to the number of floating point operations.

(i) Karmarkar's algorithm requires the most floating point operations. On the other hand, Barnes method is the most economical.

(ii) For small and medium size problems (n = 10, 20, 30, 40) less floating point operations are required by the infeasible than the predictor corrector algorithm.

(iii) For larger problems (n = 50 and 100), the predictor corrector algorithm outperforms the infeasible method.

The disagreement between CPU time and floating point operations is due to the way the floating point counter procedure (FLOPS) is implemented in MATLAB. FLOPS counts all arithmetic operations on real numbers as one floating point operation each, even though multiplications and divisions are more expensive than additions and subtractions. Taking that into account, one can understand how an algorithm that uses more multiplications and divisions than additions and subtractions consumes more CPU time, even if the same number of floating point operations is used.

From Table 4.2 and Figure 4.2, one can see that there is a big difference between the number of iterations required by each algorithm and the initial approximation of the solution that each method calculates. In general, less iterations are required by the predictor method, followed by the Barnes algorithm, the infeasible method, and, finally, the Karmarkar's dual algorithm. The best prediction to the solution vector is made by the Karmarkar algorithm, despite the poor convergence because of the small steps taken in each iteration. Conversely, the predictor corrector method converges in less than ten iterations even though the initial prediction is far and away the optimal.

Klee-Minty Problems

The class of problems originally proposed by Klee and Minty, (1972) is well known as linear programming problems with n variables for which the simplex method with various pivot rules takes an exponential in n number of pivots to reach the optimum. The following form credited to Avis and Chvatal, (1978) was considered in the experiments performed for this research, as well as in the work of Avis and Chvatal, (1978). Take the problem

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} \mu^{n-j} x_j, \\ \text{s.t.} \quad & 2\sum_{j=1}^{i-1} \mu^{i-j} x_j + x_j \le 1, && i = 1, 2, \ldots, n, \\ & x_j \ge 0, && j = 1, 2, \ldots, n, \end{aligned}$$

where $0 < \mu < 0.5$. The optimum solution of this problem is $x_j = 0$, ($j = 1, 2, \ldots, n-1$) and $x_n = 1$. Experiments for the cases with $\mu = 0.4$ and $n = 10, 20, 30, 40, 50$, and 100 were performed . The results are shown in Table 4.3. Figures 4.3 and 4.4 depict the convergence of the algorithms for one of the problems ($n = 50$) and the relative performance of the methods for problems of different size, respectively.

It was observed that, for this class of relatively sparse problems (density 50-60%), the infeasible approach stands out as the clear winner followed by the Barnes algorithm. The predictor corrector method performs better than the Karmarkar dual variant and only for really large problems ($n = 100$) does the latter shows its positive features.

Comparing the number of floating point operations, it can be seen that the Barnes algorithm uses slightly fewer operations but that is explainable by the reasons presented in the previous section.

Excluding the Karmarkar's algorithm, almost the same number of iterations was required for the remaining three methods to converge. This is a good indication of how expensive each iteration is in these three different approaches, in respect to both CPU time and the number of floating point operations.

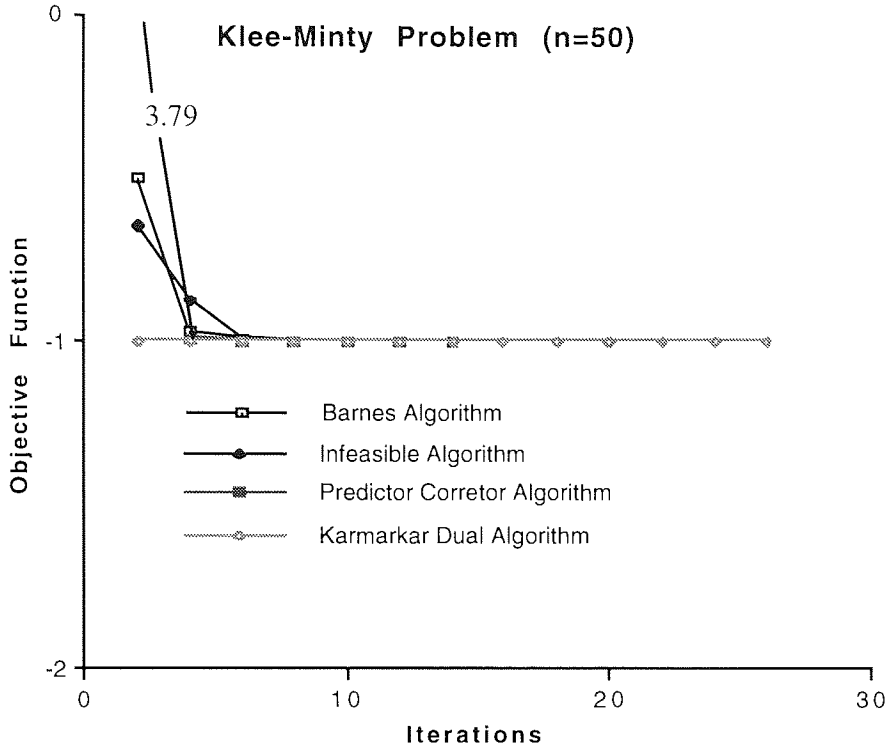

Figure 4.3 Change of the Objective Function for Klee-Minty Problems

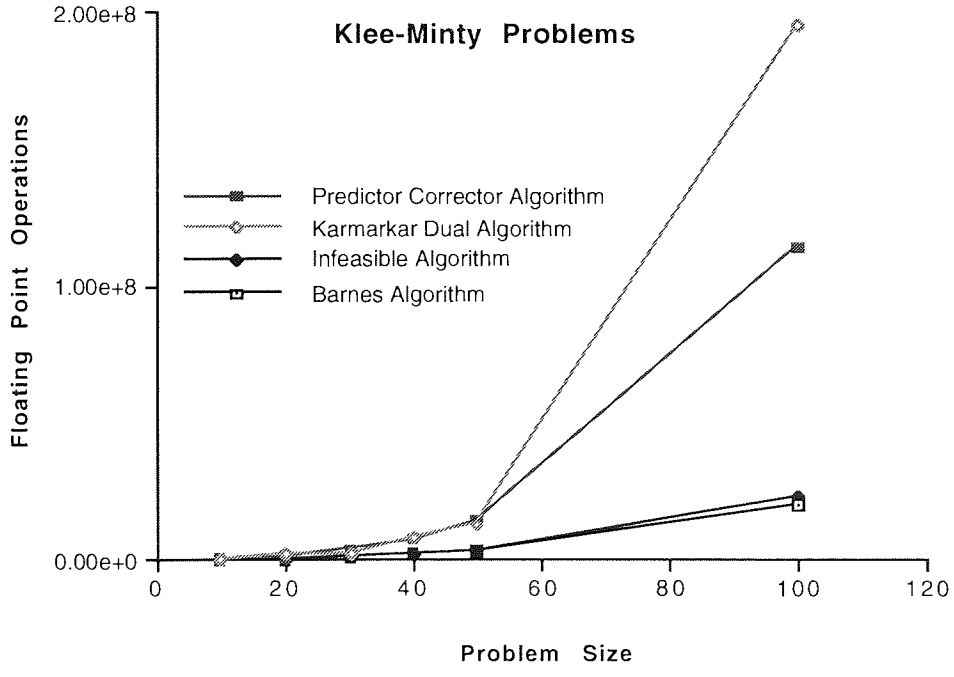

Figure 4.4 Relative Performance for Klee-Minty Problems

Studying Figure 4.3, the conclusion is reached that, as with the Hilbert-Type problems, the best and worst initial prediction for the solution vector is made by the Karmarkar algorithm and the predictor corrector method respectively.

Problem Size	Barnes Algorithm		Infeasible Algorithm		Predict Corrector Algorithm		Karmarkar Dual Variant	
Variab	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)
20	13	0.58	12	0.29	11	1.36	54	4.06
40	14	1.23	15	0.69	13	4.46	70	10.28
60	14	2.13	15	1.23	13	9.45	39	6.13
80	14	3.68	15	2.09	13	18.57	59	18.84
100	14	4.90	15	3.16	13	30.11	52	19.63

Table 4.3 Klee-Minty Problems Results

Linear Ordering Problems

The linear ordering problem can be stated as follows: for a square matrix **A** of size n whose coefficients are real numbers find the simultaneous permutation of rows and columns of **A** with the maximum sum of strictly upper triangular coefficients of the permuted matrix. The linear ordering problem is related to the optimal triangulation problem, Grötschel, et al., (1984). The optimal triangulation problem is NP-hard, [Garey & Johnson, (1979)]. Modelling the linear ordering problem as an integer linear program produces a problem with a large column to row ratio. In the experiments performed for this research, the following form, due to Karmarkar and Ramakrishnan, (1991), was considered.

$$
\begin{aligned}
\text{Max} \quad & \sum_{1\le i,j\le n} a_{ij}x_{ij} & & \\
\text{s.t.} \quad & x_{ij} + x_{ji} = 1, & & i \neq j,\ 1\le i \le n,\ 1 \le j \le n,\\
& x_{ij} + x_{jk} + x_{ki} \le 2, & & 1 \le i < j < k \le n,\\
& x_{ji} + x_{ik} + x_{kj} \le 2, & & 1 \le i < j \le k \le n,\\
& x_{ij} \text{ is 0 or 1}, & & i \neq j,\ 1\le i \le n,\ 1 \le j \le n.
\end{aligned}
$$

The problem as defined above has $n(n-1)$ variables and $n(n-1) + \frac{1}{3}n(n-1)(n-2)$ constraints. The statistics for the solved problems of this class are recorded in Table 4.4.

Problem	Original Form		Standard Form			
	Rows	Cols	Rows	Cols	Nonzeros	Density
Lnord1 (n = 3)	6	8	6	14	29	22.65 %
Lnord2 (n = 4)	12	20	12	32	74	10.57 %
Lnord3 (n = 5)	20	40	20	60	150	5.86 %
Lnord4 (n = 6)	30	70	30	100	265	3.60 %
Lnord5 (n = 7)	42	112	42	154	427	2.38 %
Lnord6 (n = 8)	56	168	56	224	644	1.65 %
Lnord7 (n = 9)	72	240	72	312	924	1.20 %

Table 4.4 Linear Ordering Problems Statistics

From the experience gathered on this class of problems, where density decreases with the size of the problem, the following conclusions can be drawn

(i) The infeasible method is the best performing algorithm among the methods tested.

(ii) The efficiency of the Karmarkar variant increases with the sparsity of the solved problem. For the most sparse of the problems tested, this algorithm is outperformed only by the infeasible method.

(iii) For very sparse problems (density less than 5%), the predictor corrector method has a very poor performance.

(iv) All implemented methods use the same number of iterations to converge, regardless of the size of the problem.

The demand on floating point operations by the tested algorithms depends on the sparsity of the problems solved, as one can see from Figure 4.6.

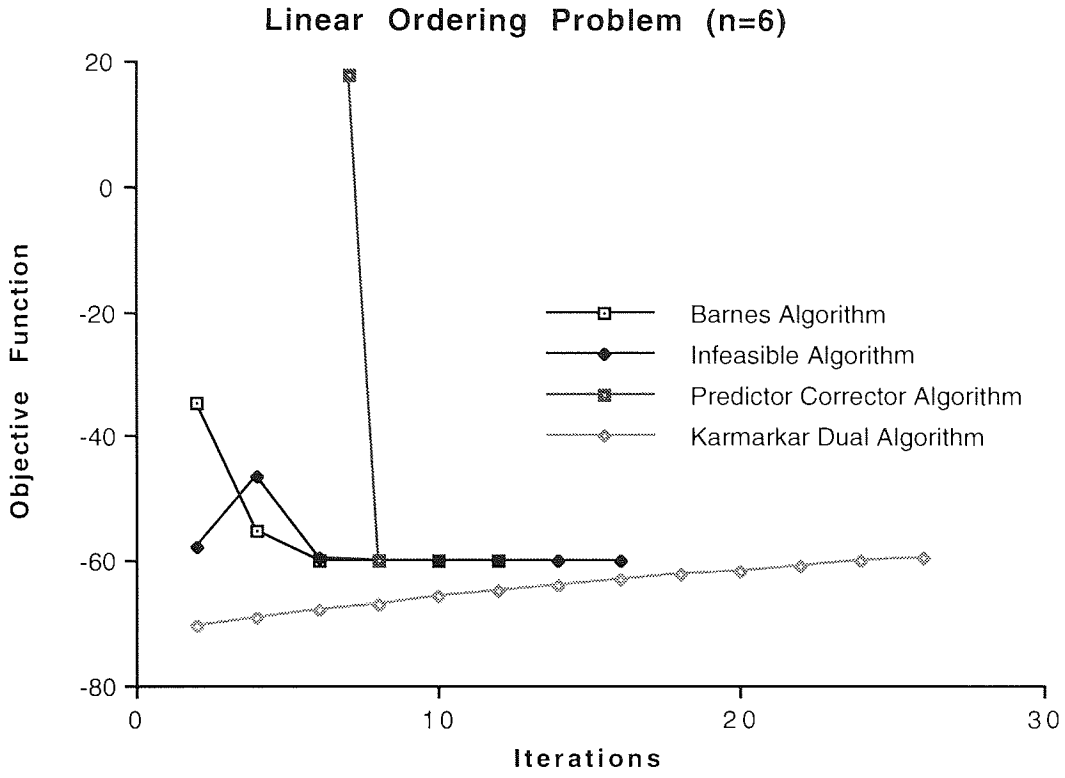

Figure 4.5 Change of the Objective Function for Linear Ordering Problems

(i) The predictor corrector method is the most demanding among the tested algorithms (Figure 4.6).

(ii) The infeasible method seems to be the clear winner for this set of sparse test problems (Table 4.5).

(ii) For problems with density less than 3%, the Barnes method is outperformed by the Karmarkar dual variant, even though the algorithm was applied on the original problem using both primal and dual phases (Table 4.5, Figure 4.6).

The convergence of the algorithms for a relatively sparse LP (density 3.60%) of this class of test problems is graphically presented in Figure 4.5. The predictor corrector method converges faster than all the rest algorithms, followed by the Barnes method. Again the Karmarkar algorithm requires many more iterations to converge to the prescribed accuracy. The best initial prediction of vector **x** was that of the infeasible method.

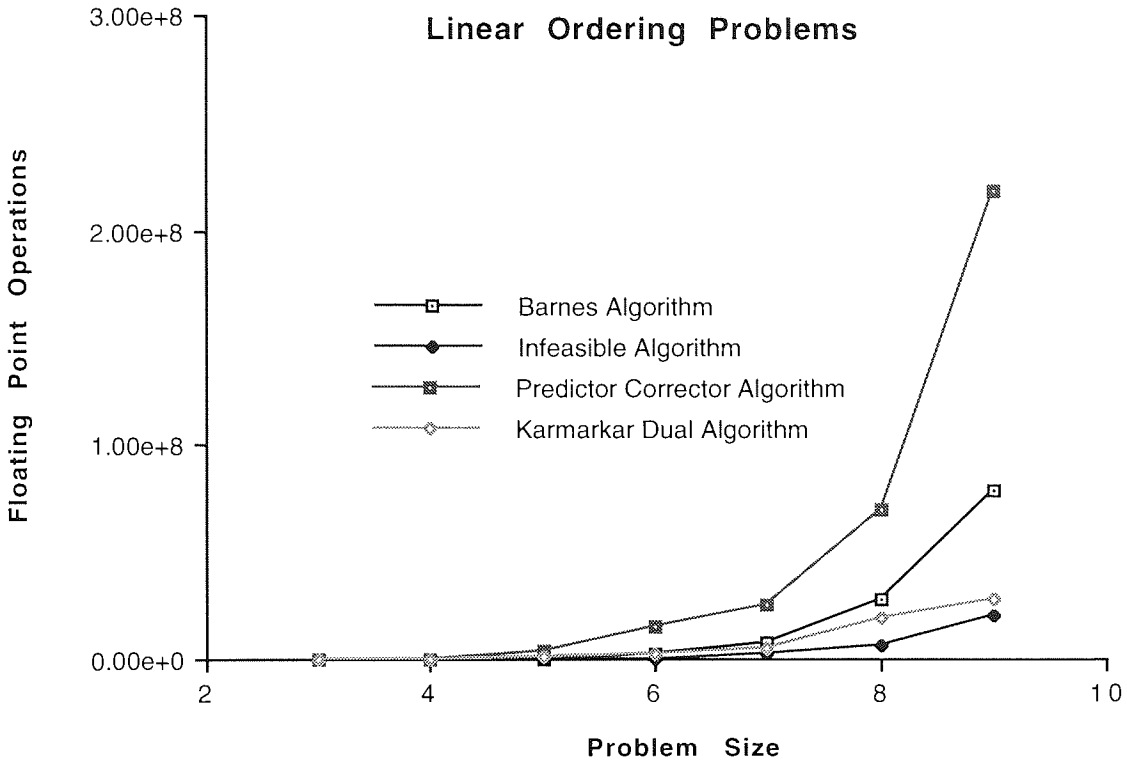

Figure 4.6 Relative Performance for Linear Ordering Problems

Problem Size	Barnes Algorithm		Infeasible Algorithm		Predictor Corrector Algorithm		Karmarkar Dual Variant	
Variab	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)
14	13	0.4	12	0.23	8	0.68	47	3.54
32	13	1.08	14	0.48	7	1.47	41	4.79
66	14	2.62	15	0.87	9	6.15	41	9.15
100	13	5.59	16	1.86	8	17.20	49	16.69
154	14	15.60	17	3.63	9	32.73	46	27.92
224	15	35.87	15	6.74	9	70.42	49	51.42
312	15	77.29	17	16.29	9	185.88	47	71.90

Table 4.5 Linear Ordering Problems Results

Random Generated Problems

The last class of test problems includes random generated linear programming problems. Both the cost vector (**c**) and the constraint matrix (**A**) are fully dense and were randomly generated in the range [0, 10] using the RAND function of MATLAB. To ensure that the problem had at least one feasible solution, the right-hand side was generated as

$$b(i) = \sum_{j=1}^{m} A(i,j), \qquad i = 1, \ldots, n.$$

Problems generated this way have at least one feasible solution, namely the unity vector. The results obtained from this set of test problems are recorded in Table 4.6. The graphical representation of the convergence of the algorithms for a random generated problem with 100 variables can be seen in Figure 4.7. Figure 4.8 depicts the number of floating point operations required by each method against the size of the problems solved.

Problem Size	Barnes Algorithm		Infeasible Algorithm		Predict Corrector Algorithm		Karmarkar Dual Variant	
Variab	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)	ITERS	CPU(s)
20	2	0.19	23	1.09	3	0.49	133*	36.60
40	2	0.62	22	3.77	3	1.68	63*	48.475
60	2	1.51	22	9.94	3	4.55	69*	122.69
80	2	3.10	19	17.56	3	8.14	86*	275.50
100	2	6.16	22	34.34	3	14.43	56*	259.45

Table 4.6 Random Generated Problems Results

The asterisks in Table 4.6 indicates that the results obtained are not optimal.

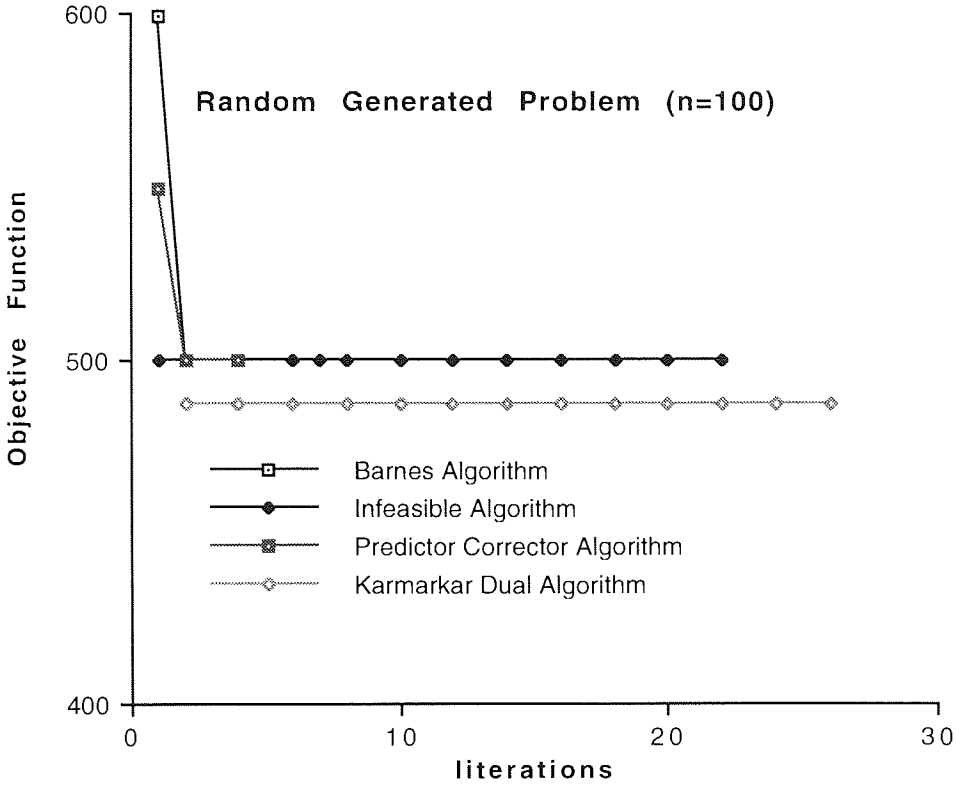

Figure 4.7 Change of the Objective Function

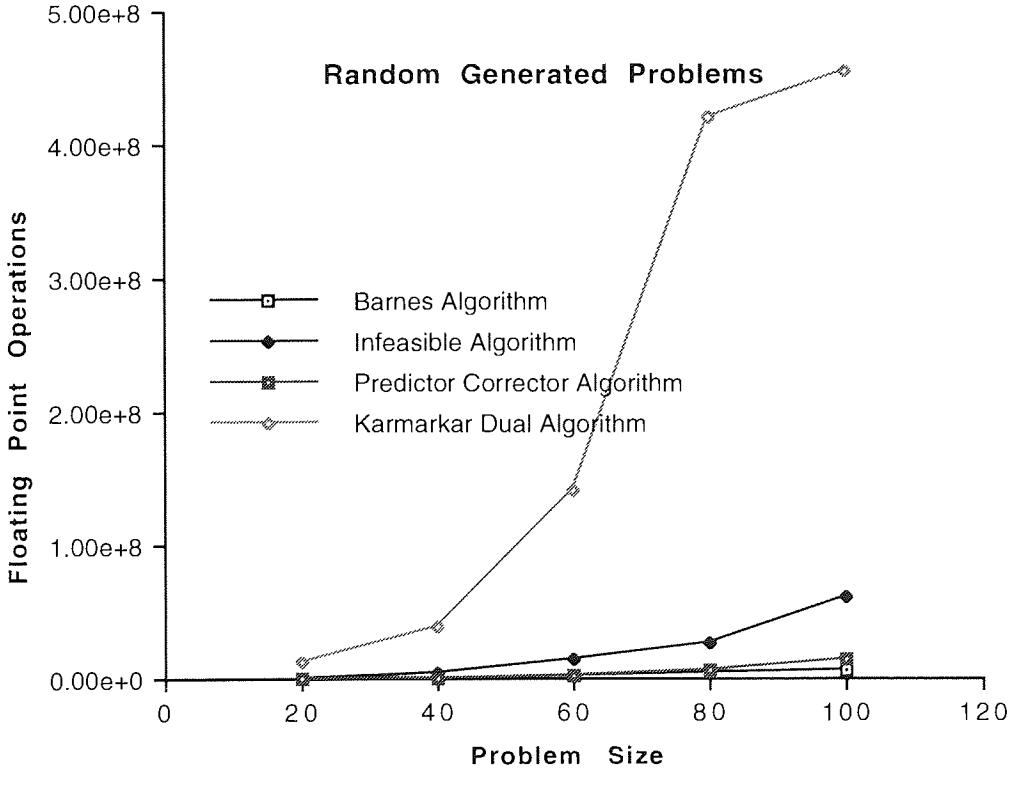

Figure 4.8 Relative Performance for Random Generated Problems

From Table 4.6 and Figures 4.7 and 4.8 the following observations are apparent

(i) The solution vector generated by the Karmarkar algorithm is the less accurate and, as one can see from Figure 4.7, the value of the objective function corresponding to that vector hardly agrees with the value obtained by the rest of the methods.

(ii) The Barnes method outperforms all the other tested algorithms even if, as seen in Figure 4.8, its initial prediction for the solution of the problem is the worst.

(iii) Again the most demanding among the tested methods, both in CPU time and number of floating point operations, is the Karmarkar algorithm.

(iv) The performance of the infeasible method is surprisingly poor for this class of test problems. That maybe is due to the fact that all the randomly generated test problems solved were fully dense.

4.7 Conclusions

As mentioned in Marsten, et al., (1990), "Interior point methods are the right way to solve large linear programs. They are also much easier to derive, motivate, and understand than they at first appeared".

In this chapter four different approaches based on interior point methods were described. Additionally, implementations of these algorithms were used for the solution of four classes of linear programming problems.

Throughout the experiments of the previous section, it was confirmed that interior point algorithms preserve their attractive features on various types of problems. These features are their low iteration count (logarithmic in the size of the problem) and their acceptance and use of the duality aspects of linear programming.

From the results recorded in Tables 4.2, 4.3 and 4.5 one can conclude that, among the tested algorithms, the cheapest iterations are those belonging to the infeasible method, followed by the Barnes algorithm, the Karmarkar variant, and, finally, the predictor corrector method. For very sparse problems, the iterations in the Karmarkar dual algorithm

become cheaper and the whole method becomes more attractive for the solution of large-scale problems. Table 4.6 shows that these observations are also valid for randomly generated problems except that now the iterations of the predictor corrector method are cheaper than these of the Karmarkar dual algorithm.

In general, the infeasible method seems to perform better than the other algorithms on most of the non-random generated problems. This is especially the case for the class of sparse problems tested where very large savings were recorded. For these problems, the infeasible algorithm required only 20% of the CPU time used by the fastest of the alternative methods. For random generated problems the Barnes algorithm stands out as the clear winner.

The small number of iterations required for the convergence of the predictor corrector method and its stability as a process indicate that the algorithm is promising but that further research is necessary for all its interesting features to be fully understood.

Chapter 5

Convex Quadratic Programming

5.1 Introduction

Quadratic programming (**QP**) is the name given to the problem of optimizing a quadratic objective function subject to linear constraints. Thus, the only difference between **QP** problems and a linear programming problem is that some of the terms in the objective function involve the square of a variable or the product of two variables, i.e., terms of the form x_j^2 and $x_j x_k$, $(j \neq k)$, [Hellier & Lieberman, (1986)]. Although **QP** could be both convex and non-convex, in this study only the convex case will be examined since only minimization problems are considered. Convex quadratic programming is an important topic in mathematical programming and it is central to many algorithms for solving non-linear programming problems. Applications of convex **QP** appear in diverse areas of engineering, mathematical, physical, social, and management sciences, [Lin & Pang, (1987)]. In the last few years quadratic programming also appears as part of decomposition algorithms for linear programming. Based on this information, one must have a good quadratic solver in hand to ensure the efficient implementation of the decomposition algorithms. Currently there is no universal "best" method for the solution of the diverse range of **QP** problems and hence many methods have been suggested for the solution of different types of quadratic programming problems. This chapter concentrates on medium and large-scale convex quadratic programming problems. After a survey of recent literature published in this area of algorithms was conducted, three methods that looked promising were selected. The evaluation and some results on small problems for the first two algorithms considered can be found in Burrett, (1994). In the implementation of the algorithms studied in this chapter, the

same optimality criterion as that in Burrett was used but parameters were adjusted in a different way to improve performance. In the conjugate gradient approach, for example, the penalty parameter is scaled by a different factor. To prevent this parameter from taking unreasonably small values, a lower bound is also set. Sparsity was also taken into account. A detailed study of the influence of sparsity and ill-conditioning on the performance of these algorithms for quadratic programming was made. These methods were then compared and their relative performances evaluated to determine which, if any, was the better method, and show situations where one algorithm out-performed the other. These investigations also served to evaluate the overall effectiveness of each algorithm.

5.2 Methods for Solving Quadratic Programs

Over the years, a large number of methods have been developed for solving convex quadratic programs. These methods can be divided into two categories: *Finite Methods* and *Iterative Methods.*

5.2.1 Finite Methods

For the solution of a program by finite methods some kind of pivoting procedure is used and termination of the process is guaranteed in finite time. Although effective for small-to-medium sized problems, finite methods tend to become less efficient and uneconomical as the problem size increases. This is due to two reasons. The first is the numerical difficulties which often occur as a result of the rapidly accumulated round-off errors. The other reason is the severe limit on the size of the problems that could be solved by these methods because of the huge computer storage required, [Lin & Pang, (1987)]. The latter is due to the fact that most finite methods operate on a certain linear complementarity problem which is derived from the optimality conditions of the program and its size is greater than that of the original

problem. In fact the number of variables in the complementarity problem is equal to m+n, where n is the number of variables in the quadratic problem and m is the number of the constraints. Finite methods are often not directly applicable to a given problem due to the changes necessary to transform the problem to the required form (refer to Pang, (1983) for a survey of finite methods for solving general convex quadratic programs).

5.2.2 Iterative Methods

Iterative methods are immune from these two handicaps because they

(i) are self-correcting and

(ii) operate on the input data only.

As a result, they are capable of preserving any data sparsity and, thus, are particularly attractive for solving large-scale sparse problems, [Lin & Pang, (1987)]. These methods generate an infinite sequence which converges to a limit point that solves the program.

5.3 QP Problem Statement and Notation

As in the linear case, there are three general formulations of the quadratic problem,

The Canonical Form :

$$\begin{aligned} &\textbf{Min (or Max) } q(x) = \tfrac{1}{2} x^T Q x + c^T x \\ &\textbf{s.t. } Ax \geq b \qquad A \in R^{m \times n},\ b \in R^m \\ &\qquad x \geq 0 \qquad x \in R^n \end{aligned}$$

The Standard Form :

$$\begin{aligned} (QP) \qquad &\textbf{Min (or Max) } q(x) = \tfrac{1}{2} x^T Q x + c^T x \\ &\textbf{s.t. } Ax = b \qquad A \in R^{m \times n},\ b \in R^m \\ &\qquad x \geq 0 \qquad x \in R^n \end{aligned}$$

The Mixed Form :

$$\begin{aligned} &\textbf{Min (or Max) } q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ &\text{s.t. } \mathbf{A}_1\mathbf{x} \geq \mathbf{b}_1 \qquad \mathbf{A}_1 \in R^{m_1 \times n},\ \mathbf{b}_1 \in R^{m_1} \\ &\quad \mathbf{A}_2\mathbf{x} = \mathbf{b}_2 \qquad \mathbf{A}_2 \in R^{m_2 \times n},\ \mathbf{b}_2 \in R^{m_2} \\ &\quad \mathbf{x} \geq \mathbf{0} \qquad \mathbf{x} \in R^n \end{aligned}$$

where $\mathbf{Q}$ is symmetric, $\mathbf{c} \in R^n$, $\mathbf{q(x)}$ denotes the quadratic objective function and T stands for transposition. The matrix $\mathbf{Q}$ is assumed to be negative definite if the problem is maximization, and positive definite if the problem is minimization. This means that $\mathbf{q(x)}$ is strictly convex in $\mathbf{x}$ for minimization and strictly concave for maximization. The linearity of the constraints guarantees a convex solution space, [Taha, (1992)].

5.4 Duality

The aim of duality is to provide an alternative formulation of a mathematical programming problem which is more convenient, computationally, or has some theoretical significance.

As in the linear case, the primal and the dual problems are related such that the Lagrange multipliers of the primal problem are part of the solution of the dual, and the Lagrange multipliers of the dual are contained in the solution of the primal.

If the original (primal) quadratic program is given in its canonical form

$$\begin{aligned} &\textbf{Min } q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ &\text{s.t. } \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ &\quad \mathbf{x} \geq \mathbf{0},\ \mathbf{x} \in R^n \end{aligned}$$

then the corresponding dual problem is

$$\begin{aligned} &\textbf{Max } \Psi(\mathbf{y}, \mathbf{w}) = \mathbf{b}^T\mathbf{y} - \frac{1}{2}\mathbf{w}^T\mathbf{Q}\mathbf{w} \\ &\text{s.t. } \mathbf{A}^T\mathbf{y} = \mathbf{Q}\mathbf{w} + \mathbf{c} \\ &\quad \mathbf{y} \geq \mathbf{0},\ \mathbf{y}, \mathbf{w} \in R^n \end{aligned}$$

5.5 Separable Quadratic Programming Problems

General, separable programming assumes that the objective function of an optimization problem is concave for a maximization problem (convex for a minimization), that each of the constraint functions is convex, and that all these functions are separable functions, [Hellier & Lieberman, (1986)]. A function $f(x_1, x_2, ..., x_n)$ is separable if it can be expressed as the sum of n single-variable functions $f_1(x_1)$, $f_2(x_2)$, ..., $f_n(x_n)$, [Taha, (1992)], that is,

$$f(x_1, x_2, ..., x_n) = f_1(x_1) + f_2(x_2) + ...+ f_n(x_n).$$

An example is the following quadratic function

$$h(x_1, x_2, ..., x_n) = a_1x_1^2 + a_2x_2^2 + ... + a_nx_n^2 ,$$

where the a_j are constants and all linear functions are separable. On the other hand, the function

$$h(x_1, x_2, ..., x_n) = a_1x_1^2 + a_2x_1x_2 + a_3x_2^2 + ... + a_nx_n^2$$

is not separable.

In the special case of quadratic programming, the constraint functions satisfy both restrictions. The quadratic objective function is said to be separable if there are no cross-product terms of the form x_jx_k, $(j\neq k)$. The objective function in a separable quadratic problem can be expressed as a sum of quadratic functions $f_j(x_j)$ of individual variables

$$\mathbf{q}(\mathbf{x}) = \sum_{j=1}^{n} f_j(x_j)$$

where each function $f_j(x_j)$ is of the form

$$f_j(x_j) = \alpha x_j^2 + \beta x_j + \gamma,$$

where α, β and γ are constants.

5.6 Sparsity and Ill-Conditioning

As in the linear case, sparsity exploitation is aimed at reducing the CPU time and storage requirements of procedures for matrix computation. In the quadratic case the main interest lies in the sparsity of the matrices **Q** and **A**. From the intensive tests carried out with different levels of density of these matrices, a huge savings was found in floating point operations and CPU time when the SPARSE option of MATLAB to declare matrix **Q** was used. Savings over the above criteria were also observed for matrix **A** in all algorithms and, especially, when the level of density was less than 10%.

Separable problems, i.e., problems with no cross-products, can be seen as the most sparse, as far as the matrix **Q** is considered, since they have non-zero entries only along the main diagonal.

To measure the ill-conditioning of the problems tested programs were implemented with different condition numbers for the matrices **Q** and **A**. The condition number of both these matrices is of significant interest in this study because of the nature of the implemented algorithms. All interior point methods involve the inverse $(\mathbf{A}\mathbf{A}^T)^{-1}$ of the matrix $\mathbf{A}\mathbf{A}^T$ in their solution process. Ill-conditioning critically affects the performance of interior point methods.

It is, therefore, essential that, for the algorithms considered in this chapter, the solution at any stage of the process be well conditioned, or, if this is not possible, at least as well conditioned as the original problem. Thus, one must try to ensure that the effect of ill-conditioning in the solution to a given problem is avoided or at least reduced as far as possible.

5.7 Model Algorithm

As stated earlier, the most effective algorithms for solving quadratic programs generate a sequence of feasible iterates. These algorithms are both practical and efficient due to their ability to develop simple characterisations of all feasible points.

Most strategies for solving a quadratic program begin with a feasible point and then seek for a feasible search direction. The algorithm then moves along this direction until either the objective function passes through a minimum or a new constraint is encountered. The process is then repeated from this new point. The following outlines a model algorithm for solving quadratic program in standard form.

Let $\mathbf{x_k}$ be the current estimate of $\mathbf{x}^*$. Given a feasible starting point $\mathbf{x_0}$, set k = 0 and repeat the following steps.

Step 1. Test convergence.

If the convergence conditions are satisfied at $\mathbf{x_k}$, the algorithm terminates with $\mathbf{x_k}$ as the solution.

Step 2. Compute a feasible search direction.

Compute a non-zero vector $\mathbf{p_k}$, the direction of search.

Step 3. Compute a step-length.

Compute a positive scalar α_k, the step-length, for which it holds that $\mathbf{F(x_k+\alpha_k p_k)<F(x_k)}$ or similar condition.

Step 4. Update the estimate of the minimum.

Set $\mathbf{x_{k+1} = x_k + \alpha_k p_k}$, k = k + 1 and go to Step 1.

Figure 5.1 shows a pictorial representation of this process.

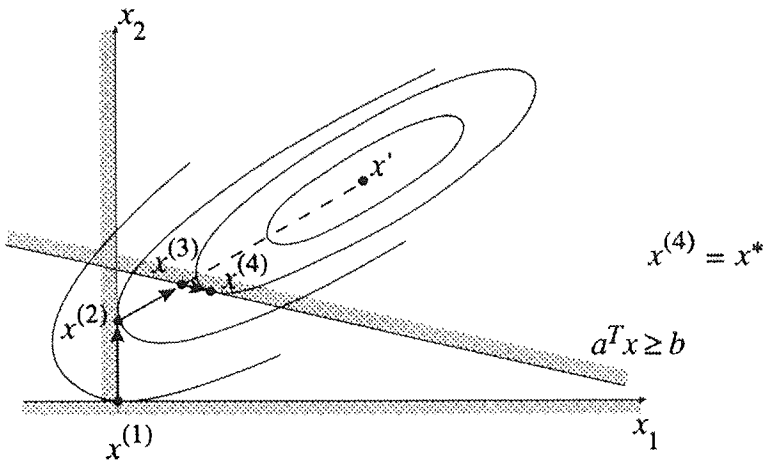

Figure 5.1 Graphical Representation of Model Method

5.8 Three Different Approaches to Quadratic Programming

In this section three different approaches for the solution of a minimization convex quadratic problem which is presented in standard form will be reviewed. One of the described methods, the unified dual ascent algorithm, handles upper bounds explicitly. This algorithm is reported to give good comparative results on separable quadratic programs. The other two methods can also handle upper bounds subject to the addition of an extra constraint and an artificial variable for each bounded variable.

For the algorithms described in the following sections it is assumed that the problem is feasible and the quadratic coefficient matrix **Q** is positive definite.

5.8.1 The Algorithm of T.J. Carpenter and D.F. Shanno

In Chapters 3 and 4 it was explained how conjugate gradient methods work and it was shown how that powerful tool is used in linear programming. The conjugate direction methods are notable by their key property: that they minimize a positive definite quadratic

function in n or less steps. Conjugate gradient methods are particularly suited to large-scale problems because they generate directions of search without storing a matrix. Therefore, this class of algorithm is necessary in situations where matrix factorization type methods are not viable due to the size or density of the relevant matrices.

In the current section a quadratic programming algorithm based on the same principles will be described. The method is a doubly iterative algorithm for problems in standard form, described in detail in Carpenter and Shanno, (1993), that works in the null space of **A.** The search directions are obtained with the use of a conjugate projected procedure. The main advantage of this approach is that **Q** appears in a conjugate direction routine rather than in a matrix factorization. This method considers the primal quadratic program **QP** and its associated barrier transformation

$$\begin{aligned} &\textbf{Min } F(x) = \tfrac{1}{2} x^T Q x + c^T x - \mu \sum_{j=1}^{n} \ln(x_j) \\ &\text{s.t. } Ax = b \\ &\qquad x \geq 0 \end{aligned}$$

Henceforth this will be referred to as **BP**.

To solve **QP** by the logarithmic barrier function method one approximately solves a sequence of problems **BP** where the positive parameter μ_k is decreased as k is increased so that $\mu_k \rightarrow 0$ as $k \rightarrow \infty$. To approximately solve **BP**(μ_k), one can use any appropriate algorithm for the solution of such problems, starting from the approximate optimal solution x^k of the previous problem **BP**(μ_{k-1}). If μ_k is close to μ_{k-1} and x^k is close to x^k_{opt}, the optimal solution of **BP**(μ_{k-1}), then x^{k+1}_{opt}, should not be too distant from x^k. Hence, it should not be too difficult to compute a good approximation x^{k+1} to x^{k+1}_{opt} starting from x^k. The next point x^{k+1} is then computed from x^k by taking a single Newton step Δx^k, so $x^{k+1} = x^k + \Delta x^k$. To obtain Δx^k, one must solve the equality constrained quadratic program

$$\begin{aligned} &\textbf{Min } \tfrac{1}{2} \Delta x^T Q_k \Delta x + q^{(k)T} \Delta x \\ &\text{s.t. } A\Delta x = 0, \end{aligned}$$

where $\mathbf{Q_k} = \mathbf{Q} + \mu_k \mathbf{X}_k^{-2}$ and $\mathbf{q^k} = \mathbf{Qx^k} + \mathbf{c} - \mu_k \mathbf{X}_k^{-1}\mathbf{e}$. The objective function of this quadratic program is the second order Taylor series expansion of the objective function of **BP** about the feasible point $\mathbf{x^k}$. Henceforth this will be referred to as $\mathbf{QP}(\mu_k,\mathbf{Q_k})$.

Note that the sufficient condition for the Newton step $\Delta\mathbf{x}^* = -[\nabla^2 QP(\mu_k,\mathbf{Q_k})(\mathbf{x^k})]^{-1} \nabla QP(\mu_k,\mathbf{Q_k})(\mathbf{x^k})$ to be optimal in $\mathbf{QP}(\mu_k,\mathbf{Q_k})$ is that the gradient of $\mathbf{QP}(\mu_k,\mathbf{Q_k})$ is equal to zero and the projection of the gradient in this space is zero as well. This algorithm uses the projection matrix

$$\mathbf{P} = \mathbf{I} - \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{A},$$

for which optimality requirements are described as

$$\mathbf{P}(\mathbf{Q_k}\Delta\mathbf{x}^* + \mathbf{q}^{(k)}) = \mathbf{0} \qquad (5.8.1)$$

The solution of problem (5.8.1) is obtained using a conjugate direction procedure. The conjugate projected gradient method can therefore be viewed as a conjugate direction procedure applied directly to problem (5.8.1).

The proposed algorithm remains viable even when $\mathbf{Q_k}$ is large or dense because it applies a conjugate direction procedure to solve $\mathbf{QP}(\mu_k,\mathbf{Q_k})$. However, the matrix that appears in this system is not necessarily symmetric or positive definite; therefore, it is not immediately apparent that a conjugate direction method is applicable. In Carpenter and Shanno, (1993) it is proved that the proposed method is in fact a conjugate direction routine that converges in, at most, n–m iterations. The conjugate projected gradient method is developed by applying the standard conjugate gradient method to solve an unconstrained quadratic program that is equivalent to $\mathbf{QP}(\mu_k,\mathbf{Q_k})$. For more details about that refer to Carpenter and Shanno, (1993).

The algorithm can be described step by step as follows:

While $\nabla F(\mathbf{x}^k) \neq 0$

Step 0. Start at $\Delta \mathbf{x}_0 = \mathbf{0}$ and define (a) $\mathbf{q}_0 = \mathbf{q}$ and (b) $\mathbf{d}_0 = -\mathbf{P}\mathbf{q}_0$. (c) Set i = 0.

Step 1. **While** $Pq_i \neq 0$

(a) Compute $\gamma_i = -\dfrac{\mathbf{d}_i^T \mathbf{q}_i}{\mathbf{d}_i^T \mathbf{Q}_k \mathbf{d}_i}$.

(b) Let $\Delta \mathbf{x}_{i+1} = \Delta \mathbf{x}_i + \gamma_i \mathbf{d}_i$.

(c) Set $\mathbf{q}_{i+1} = \mathbf{Q}_k \Delta \mathbf{x}_{i+1} + \mathbf{q}_i$ and $\beta_i = \dfrac{\mathbf{d}_i^T \mathbf{Q}_k \mathbf{q}_{i+1}}{\mathbf{d}_i^T \mathbf{H} \mathbf{d}_i}$.

(d) Obtain a new direction $\mathbf{d}_{i+1} = -\mathbf{P}\mathbf{q}_{i+1} + \beta_i \mathbf{d}_i$.

(e) Update counter i= i +1.

end

Step 2. Assign $\Delta \mathbf{x} = \Delta \mathbf{x}_i$ and **stop.**

(a) Set $\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta \mathbf{x}$

(b) k = k + 1

end

For the algorithm stated above it is assumed that at the initial point $\mathbf{x}_0$, $\Delta \mathbf{x}_0 = \mathbf{0}$. In practice, when $\mathbf{Q}_k \in R^{n \times n}$ any $\Delta \mathbf{x}_0 \in R^n$ may be selected.

5.8.2 Goldfarb and Lui $O(n^3L)$ Primal Interior Point Algorithm

As described in Goldfarb and Lui, (1991), this is an interior point method for convex programming which is based upon a logarithmic barrier function approach.

The algorithm generates a sequence of problems, each of which is approximately solved by taking a single Newton step. In Goldfarb and Lui, (1991) it is also shown that the method requires $O(\sqrt{n}L)$ iterations and $O(n^{3.5}L)$ arithmetic operations. By using modified Newton steps the number of arithmetic operations required by the algorithm can be reduced to $O(n^3L)$. To maintain primal and dual feasibility scaling the current primal solution is employed.

The initial steps of this algorithm are identical to that of the approach described in the previous section. Each step of the algorithm is determined by applying Newton's method directly to the barrier function minimization problem instead of to a non-linear system of

equations that is equivalent to the Karush-Kuhn-Tucker optimality conditions for a minimization problem corresponding to the current value of the logarithmic barrier parameter. Starting at a suitable interior point and then taking suitably small steps, the Newton barrier function generates dual as well as primal feasible solutions. The duality gap corresponding to these solutions is driven to zero at a fixed rate of $1 - \sigma/\sqrt{n}$, where σ is a given, positive constant.

This method again considers the primal quadratic program **QP** and its associated barrier transformation **BP** referred to in the previous section. If one assumes that

(i) **QP** has a strictly positive feasible solution;

(ii) the set of optimal solutions of **QP** is non-empty and bounded;

then each problem **BP** has a optimal solution $\mathbf{x}(\mu)$ and it can be shown that $\mathbf{x}(\mu) \rightarrow \mathbf{x}^*$, the optimal solution of **QP**, as $\mu \rightarrow 0$.

The algorithm actually uses a "modified" Newton method for approximately solving **BP**. Specifically, it determines the step $\Delta\mathbf{x}^k$ by solving $\mathbf{QP}(\mu_k,\mathbf{Q'}_k)$, where

$$\mathbf{Q'}_k = \mathbf{Q} + \varepsilon_k \mathbf{Z}_k^{-2},$$

$\mathbf{z}^k$ is close to $\mathbf{x}^k$, ε_k is close to μ_k, $\mathbf{z}^k > \mathbf{0}$, and $\varepsilon_k > 0$ so that $\mathbf{Q'}_k$ is positive definite. If $\Delta\mathbf{x}^k$ is the optimal solution to $\mathbf{QP}(\mu_k,\mathbf{Q'}_k)$, then a vector $\mathbf{y}^{k+1} \in R^m$, of Lagrange multipliers exists that

$$\mathbf{Q'}_k\Delta\mathbf{x}^k + \mathbf{q}^k - \mathbf{A}^T\mathbf{y}^{k+1} = \mathbf{0}.$$

Under the assumption that rank($\mathbf{A}$) = m, $\Delta\mathbf{x}^k$ can be written explicitly as

$$\Delta\mathbf{x}^k = \mathbf{Q'}_k^{-1}(\mathbf{A}^T\mathbf{y}^{k+1} - \mathbf{q}^k),$$

where

$$\mathbf{y}^{k+1} = (\mathbf{A}\mathbf{Q'}_k^{-1}\mathbf{A}^T)^{-1}\mathbf{A}\mathbf{Q'}_k^{-1}\mathbf{q}^k.$$

Using the "modified" Newton method, i.e., by not changing ε_k and all components in $\mathbf{x}^k$ in the definition of $\mathbf{Q}_k$ from one iteration to the next, it is proved to reduce the amount

of computation needed on the average at each iteration by a factor of $O(\sqrt{n})$ without increasing the order of the number of iterations required by the algorithm, [Goldfarb & Lui, (1991)]. The idea of using "modified" projections to reduce the work per projection over all iterations was first introduced by Karmarkar, (1984) and it is based on allowing μ_k to change at most O(L) times during the course of $O(\sqrt{n}L)$ iterations, where n is the number of variables and L is the input length of the quadratic program. That means that $\mathbf{Q'_k}$ differs from $\mathbf{Q_k}$ by, at most, $O(\sqrt{n})$ diagonal elements.

To begin the algorithm, one needs to transform the **QP** problem into a quadratic program that satisfies assumptions (i), (ii), and has $\mathbf{x_0} = \mathbf{e}$ ($\mathbf{e}$ is equal to a suitable length vector of ones) as a feasible point, where $\mathbf{x_0}$ satisfies $\mathbf{x_0 A^T y} = \mathbf{e}$, for some $\mathbf{y} \in R^m$. The proposed approach is one that has been suggested by numerous authors for transforming linear programs into a form suitable for interior point algorithms. It consists of introducing two new variables and associate large positive constants as penalty parameters to them.

If the penalty parameters are large enough and if **QP** has optimal solution then the transformed problem will have optimal solutions $\mathbf{x}^*$ with $x^*_{n-1} = 0$ and an optimal solution of **QP** can be obtained by using the first n–2 components of $\mathbf{x}^*$. If **QP** is infeasible x^*_{n-1} will be positive. A step by step description of the algorithm is as follows

Step 0. Let (a) $\mathbf{x_0} > \mathbf{0}$ be a given feasible point for a quadratic problem in standard form and let (b) $\mu_0 > 0$, $\tau > 0$, $\sigma > 0$ and $\gamma \geq 0$ be given constants. (c) Set k = 0.

Step 1. Choose (a) $\mathbf{z^k}$ and (b) ε_k that satisfy

$$\frac{|x_i^k - z_i^k|}{z_i^k} \leq \gamma, \; i = 1, 2, \ldots, n,$$

and

$$\frac{|\mu_k - \varepsilon_k|}{\varepsilon_k} \leq \gamma.$$

Step 2. (a) Compute $\Delta\mathbf{x}$, the solution of $\mathbf{QP}(\mu_k, \mathbf{Q'_k})$, where

$\mathbf{Q'_k} = \mathbf{Q} + \varepsilon_k \mathbf{Z_k^{-2}}$, as described above,

set

$$\text{(b) } \mathbf{x}^{k+1} = \mathbf{x}^k + \Delta\mathbf{x}^k,$$

and (c) compute the duality gap $(\mathbf{x}^{k+1})^T\mathbf{s}^{k+1}$, where $\mathbf{s}^{k+1} = \mathbf{Q}\mathbf{x}^{k+1} + \mathbf{c} - \mathbf{A}^T\mathbf{y}^{k+1}$ and $\mathbf{y}^{k+1}$ is the optimal vector of Lagrange multipliers for $\mathbf{QP}(\mu_k, \mathbf{Q'}_k)$.

Step 3. If $(\mathbf{x}^{k+1})^T\mathbf{s}^{k+1} < \tau$ then, STOP; else, (a) set $\mu_{k+1} = (1 - \sigma/\sqrt{n})\mu_k$ and (b) k = k+1 and go to Step 1.

In Step 1 the components of vector $\mathbf{z}^k$ are chosen to be close enough, to a tolerance γ, to these of vector $\mathbf{x}^k$ and ε_k close to μ_k. That allows ε_k to be set to μ_k, at most, once every $\gamma\sqrt{n}\sigma$ iterations and the algorithm to use "modified" Newton steps which reduce the number of arithmetic operations required by a factor of $O(\sqrt{L})$.

Parameters γ and σ are selected to satisfy the inequalities

$$0 < \delta < \left(\frac{1-\gamma}{1+\gamma}\right)^3 + \frac{1-\gamma}{(1+\gamma)^2} - 1,$$

$$0 < \sigma \le \frac{\left[\frac{(1-\gamma)^2\delta}{1+\gamma} + 1\right]\delta - \frac{(1+\gamma)^2}{1-\gamma}(1+\delta)\delta}{1 + \frac{(1-\gamma)^2}{1+\gamma}\frac{\delta}{\sqrt{n}}}.$$

5.8.2.1 The Role of the Barrier Parameter

When logarithmic barrier functions are used in interior point methods, [Lustig, et al., (1991); Monteino & Adler, (1989)], a close relation exists between the step-length and the barrier parameter μ. Algorithms for choosing an initial μ^0 and reducing μ at each step in order to assure polynomial convergence of barrier methods were developed first for primal algorithms by Gonzaga, (1987). Since these algorithms reduce μ by a very small multiple at each step by a formula similar to

$$\mu^{k+1} = \mu^k\left(1 - \frac{0.1}{\sqrt{n}}\right),$$

they are hopelessly slow in practice.

An alternative is to choose μ by the algorithm of McShane, et al., (1989), namely, for a pair of feasible primal and dual vectors $\mathbf{x}$ and $\mathbf{y}$,

$$\mu = (\mathbf{c}^T\mathbf{x} - \mathbf{b}^T\mathbf{y})/n^2 .$$

The above formula allows great decreases in the value of the barrier parameter when the current estimate of the solution vector is far from the optimal solution (big duality gap) and small changes when the optimal solution is approached (small duality gap). As an alternative to the argued algorithm this approach was used to reduce the barrier parameter in the experiments performed in this research and seems to work well. However, the problems considered in this chapter are quadratic so the objective functions used in the above formula were extrapolated to the quadratic case.

5.8.3 The Unified Dual Ascent Algorithm

To complete this study a totally different approach to convex quadratic programming will be described in this section. As described in Lin and Pang, (1987) this algorithm solves the primal program by generating a sequence of dual vectors $\{\mathbf{y}^k\}$, which induces the sequence of primal vectors $\{\mathbf{x}(\mathbf{y}^k)\}$, through the maximization of the dual Lagrangian function

$$\mathbf{d}(\mathbf{y}) = \mathbf{min}_{\mathbf{x}\geq 0}\ \mathbf{c}^T\mathbf{x} + \tfrac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{y}^T(\mathbf{b} - \mathbf{A}\mathbf{x}),$$

which is unrestricted over $\mathbf{y}$.

The generation of $\mathbf{y}^k$ is as follows. The initial $\mathbf{y}^0$ is arbitrary. In general, given $\mathbf{y}^k$, choose a search direction δ^k. Define $\mathbf{y}^{k+1} = \mathbf{y}^k + \theta^k\delta^k$ where θ^k is such that

$$\mathbf{d}(\mathbf{y}^{k+1}) = \max_{\theta} \mathbf{d}(\mathbf{y}^k + \theta\delta^k).$$

Since

$$\mathbf{d}(\mathbf{y}^k + \theta\delta^k) = \min_{x\geq 0} \mathbf{f}(\mathbf{x}) + (\mathbf{y}^k)^T(\mathbf{b} - \mathbf{A}\mathbf{x}) + \theta(\delta^k)^T(\mathbf{b} - \mathbf{A}\mathbf{x}),$$

where

$$\mathbf{f}(\mathbf{x}) = \mathbf{c}^T\mathbf{x} + \tfrac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x},$$

it follows that the search for θ^k can be found by solving the subprogram

(SQP) $\quad$ **Min** $\mathbf{f}(\mathbf{x}) + (\mathbf{y}^k)^T(\mathbf{b} - \mathbf{A}\mathbf{x})$
s.t. $(\delta^k)^T(\mathbf{b} - \mathbf{A}\mathbf{x}) = 0$ and $\mathbf{x} \geq 0$

and by letting θ^k be an optimal Lagrange multiplier of the constraint $(\delta^k)^T(\mathbf{b} - \mathbf{A}\mathbf{x}) = 0$. Note that **SQP** is a single constraint problem. The subprogram **SQP** is clearly feasible (and, thus, solvable). This follows from the assumption which was accepted earlier that the initial quadratic problem is feasible. Indeed, the unique minimizer of **SQP** is the vector $\mathbf{x}(\mathbf{y}^{k+1})$.

The constraint $(\delta^k)^T(\mathbf{b} - \mathbf{A}\mathbf{x}) = 0$ in **SQP** represents an aggregation of the constraints $\mathbf{A}\mathbf{x} = \mathbf{b}$. Thus, one may interpret the above dual ascent method as solving a sequence of simplified subproblems, each of which has the same objective function as **QP** but modified by the Lagrangian term involving the complicating constraints aggregated into a single one by a certain vector, δ^k.

Observe that $\mathbf{x}(\mathbf{y}^k)$ is obtained as a by-product of the method and no extra effort is required for its computation.

There are many choices for the aggregation vector δ^k. Two large families of such choices are (i) Gradient-type ascent, in which $\delta^k = \mathbf{H}^k(\mathbf{b} - \mathbf{A}\mathbf{x}(\mathbf{y}^k))$ where $\mathbf{H}^k$ is some symmetric matrix and (ii) Periodic basic ascent. Note that if it is supposed that **QP** is feasible, then the dual function $\mathbf{d}(\mathbf{y})$ is continuously differentiable and $\nabla\mathbf{d}(\mathbf{y}^k) = \mathbf{b} - \mathbf{A}\mathbf{x}(\mathbf{y}^k)$. Gradient-type methods include the steepest ascent method which has $\mathbf{H}^k$ equal to the identity matrix, many quasi-Newton methods, as well as the conjugate gradient method with restart that has

$$\mathbf{H}^k = \begin{cases} \mathbf{I} & \text{if } k \equiv 0 \pmod{N} \\ \mathbf{I} + \delta^{k-1}\dfrac{\nabla\mathbf{d}(\mathbf{y}^k) - \nabla\mathbf{d}(\mathbf{y}^{k-1})^T}{\nabla\mathbf{d}(\mathbf{y}^{k-1})^T\nabla\mathbf{d}(\mathbf{y}^{k-1})} & \text{otherwise} \end{cases}$$

where N is some positive integer not exceeding the number of equations m. Under the last definition, the vector δ^k is given by

$$\delta^k = \begin{cases} \nabla \mathbf{d}(\mathbf{y}^k) & \text{if } k \equiv 0 \pmod{N} \\ \nabla \mathbf{d}(\mathbf{y}^k) + \delta^{k-1} \dfrac{\nabla \mathbf{d}(\mathbf{y}^k)^T (\nabla \mathbf{d}(\mathbf{y}^k) - \nabla \mathbf{d}(\mathbf{y}^{k-1}))}{\nabla \mathbf{d}(\mathbf{y}^{k-1})^T \nabla \mathbf{d}(\mathbf{y}^{k-1})} & \text{otherwise} \end{cases}$$

which is the Polak-Ribiera-Polyak conjugate gradient formula, [Avriel, (1976)]. The integer N denotes the number of iterations after which the method is restarted with the steepest ascent direction.

The maximization problems generated by the algorithm are bounded quadratic optimization problems with one linear constraint. The solution of these problems can be obtained by any algorithm for quadratic programming. However, for the algorithm to be efficient a fast solver of this special problem is required. The best theoretical bound, for a general quadratic algorithm, obtained to date is that of $O(n^3L)$ arithmetic operations, [Carpenter & Shanno, (1993)]. However, the special form of the subproblems allows the use of specialized algorithms. A survey of literature related to these specialized algorithms was conducted and provided two promising algorithms in its conclusion. The first was an algorithm of complexity O(nlogn), [Helgason, et al., (1980)]. The main disadvantage of this method is that it requires the right hand side of the single constraint to be equal to 1. Even though any problem can be transformed in that form, scaling is required. The second method is of complexity O(n) and is immune from the above requirement, [Brucker, (1984)]. The algorithm is based on a parametric approach combined with well-known ideas for constructing efficient algorithms.

The main idea of the algorithm lies in the construction of a parametric problem with only lower and upper bounds on the variables. The optimum value of that problem is initially bracketed in the interval between the minimum and the maximum of the critical parameters of the parametric problem. The critical parameters are defined as the point between which the structure of the solution of the parametric problem does not change. At every iteration of the algorithm the set of the critical parameters reduces at least by half. Once the optimum value

of the parametric problem is found the solution of the single constraint problem is obtained by back substitution.

The unified dual ascent algorithm is particularly attractive for problems with separable objective function. If the objective is non-separable, then a transformation is used to convert it to a separable form. The transformation depends on two factors: knowledge of the smallest eigenvalue p of $\mathbf{Q}$ (or at least a lower bound) and the factorization $\mathbf{G}$ of the matrix $\mathbf{Q} - \mu\mathbf{I}$, where $0 < \mu < p$. When both μ and the factorization are computed, then $\mathbf{Q}$ can be written as $\mathbf{Q} = \mu\mathbf{I} + \mathbf{G}^T\mathbf{G}$ and as described in Lin and Pang, (1987), obviously, **QP** is equivalent to

$$\textbf{(QPS)} \qquad \textbf{Min } \mathbf{c}^T\mathbf{x} + \tfrac{1}{2}\mu\mathbf{x}^T\mathbf{x} + \tfrac{1}{2}\mathbf{y}^T\mathbf{y}$$
$$\textbf{s.t. } \mathbf{Ax} = \mathbf{b},\ \mathbf{Gx} = \mathbf{y},\ \mathbf{x} \geq \mathbf{0}.$$

The latter program has a strictly convex separable objective and can be solved with the algorithm described. In the implementation of this research the factorization was done by the CHOL procedure of MATLAB, which produces the Cholesky factor of a square matrix.

5.8.3.1 Implementation Issues

The difficulties in the implementation of the unified dual ascent method lie in the coding of the algorithm used for the solution of the single constraint quadratic problem. This is a difficult algorithm to implement efficiently and required the development of set manipulation functions which are not directly available in the MATLAB environment. These operations are necessary because in each iteration of the algorithm the difference between sets of numbers is computed, i.e., the common elements of two sets are eliminated. The sets considered are the set of variables whose value remains unspecified and the set of variables for which the optimal value is obtained in the current iteration. A procedure SETSUB which has two sets as input arguments and output their difference was implemented in MATLAB to deal with this problem.

Another interesting point in the implementation of the unified dual ascent method is that θ^k, the optimal Lagrange multiplier of the constraint in the single constraint quadratic problem, is obtained as a by-product of the algorithm with no expense.

As mentioned in the previous section, a transformation is proposed for the solution of non-separable problems. The smallest of the eigenvalues required in that transformation can be obtained using the procedure EIG of MATLAB. An iterative process can also be used for the same purpose as described in Lindfield and Penny, (1995). Both approaches are efficient for small matrices but as the size of the matrix is increased, they become more expensive. However, experimental results showed that the unified dual ascent algorithm does not perform well on the transformed problems.

5.9 Computational Experience

In the last section three iterative methods for solving convex quadratic programs were described. From a practical point of view, it is important to know how these methods perform and compare to one another. In this section the numerical results of extensive computer experiments using these methods will be reported. Data of the test problems were randomly generated and all the computations were performed on a networked SUN SPARC workstation. The computer codes were written in MATLAB, ver.4.2 double precision. Two sets of experiments were performed; one consisted of problems with separable objective functions and the other with non-separable functions.

5.9.1 Tests on Random Generated Separable QP Problems

In the first set of experiments, a separable, strictly convex, quadratic problem of the form **QP** with **Q** a positive diagonal matrix was considered. In this set of experiments, **Q** was generated by the EYE procedure of MATLAB. EYE generates identity matrices of a

given size. Sparse identity matrices were generated by the SPEYE function. To ensure that the system $\mathbf{Ax} = \mathbf{b}$ was feasible, the RHS vector $\mathbf{b}$ was generated as $\mathbf{Ax}^\circ$, where $\mathbf{x}^\circ$ is a vector of ones. To determine if the solution found was close enough to the optimal one the duality gap was used. The following termination criterion was used in all three methods:

$$\text{abs}((pv - dv)/pv) \leq 0.0005,$$

where pv is the value of the primal objective function and dv is the value of the dual. The number m of the rows in matrix $\mathbf{A}$ was chosen to be n/2, where n is the dimension of the $\mathbf{x}$-vector. Five values of n where chosen: 10,20,30,50, and 100. Matrix $\mathbf{A}$ has no specific structure and the non-zero entries are not necessarily 1. It has been found that the GL method is very sensitive to the parameters γ, δ, and σ. The method was run with different values of these parameters and the best values were chosen. The values compared were the suggested ones ($\gamma = 0.1$, $\delta = 0.1$, $\sigma = 0.023$, and $\gamma = 0$, $\delta = 0.5$, $\sigma = 0.1666$) and one that was found to perform better on most of the problems ($\gamma = 0$, $\delta = 0.5$, $\sigma = 1.5$). The results are summarized in Table 5.1. The column for the GL method in that table gives the results pertaining to these new values for the above parameters and without using the initial transformation suggested in Goldfarb and Lui, (1991), which ensures that a vector of ones is an initial feasible solution to the problem. The algorithm can be used without this initialization step since the test problems are composed in such a way that the latest requirement is already satisfied.

	CG		GL		UDA	
Size	Flops	Flops/n^3	Flops	Flops/n^3	Flops	Flops/n^3
5×10	0.305E6	305	0.147E6	147	0.028E6	28
10×20	1.720E6	215	1.644E6	205	0.226E6	28
15×30	2.706E6	100	6.198E6	229	0.757E6	28
25×50	14.100E6	112	38.686E6	309	3.570E6	28
50×100	168.560E6	168	454.846E6	454	24.293E6	24

Table 5.1 Separable Dense Problems Results

From the results in Table 5.1 the following observations are apparent

- The unified dual ascent algorithm stands out as the clear winner for this set of test problems.
- For problems with n > 10, the conjugate gradient method out-performs the GL method and, as the size of the problems increases, it tends to be most preferable.
- The GL method takes more iterations to converge but its iterations are cheaper than those of the conjugate gradient.
- In the conjugate gradient method, the number of flops depends on the sparsity of matrix **A**. On the other hand, the GL method does not seems to depend on this factor.
- All algorithms seems to keep the cost of each iteration constant, independent of the size of the problem.

Next, the same set of test problems was run with the option SPARSE. All matrices were declared as sparse, which means that the zero elements of a matrix were not stored and computations with them were not performed. This is due to that fact that a big savings was expected both in time and floating point operations, especially for very sparse problems. For each problem different values of the density of matrix **A** were tested (3%, 8%, 14%, 50%). The results for these runs are summarized in Table 5.2. Graphical representation of the convergence for all algorithms for a sparse problem with 20 variables against the number of iterations, the CPU time and the number of floating point operations are presented in Figures 5.2 through 5.4.

One can conclude from the results in Table 5.2 that:

- For very sparse problems (density 3%) the GL method stands out as the clear winner regardless of the size of the problem.
- For very dense problems (density 50%) the UDA method performs better than the other two algorithms.
- For medium size problems (n = 20, 30) and densities 8% and 14% the UDA method has a very poor performance. On the other hand the GL method needs only half the number of floating point operations required by the CG method which has the second best performance for these problems.

- For large problems (n = 50, 100) and densities 8 and 14% the UDA method has the best performance among the tested algorithms followed by the GL and the CG algorithms.

The good performance of the UDA method on large and dense problems could be due to the way it handles problems, i.e., solving a sequence of single constraint subproblems, which is cheap as far as floating point operations are concerned. The efficiency of the GL method on very sparse problems is expected because the main computational step, the solution of a system of simultaneous equations, is done by the built-in function of MATLAB, which was proved to be very efficient for sparse matrices.

Size	Dens	CG_S Flops	CG_S Flops/n^3	GL_S Flops	GL_S Flops/n^3	UDA_S Flops	UDA_S Flops/n^3
5×10							
	3%	0.481E6	48	0.010E6	10	0.012E6	12
	8%	0.379E6	38	0.010E6	10	0.011E6	12
	14%	0.801E6	80	0.033E6	34	0.029E6	29
	50%	0.124E6	125	0.025E6	25	0.023E6	23
10×20							
	3%	0.511E6	15	0.040E6	5	0.099E6	12
	8%	0.738E6	9	0.041E6	5	0.134E6	17
	14%	0.192E6	24	0.084E6	10	0.196E6	24
	50%	0.628E6	78	0.292E6	36	0.271E6	34
15×30							
	3%	0.143E6	5	0.077E6	2	0.393E6	14
	8%	0.261E6	9	0.114E6	4	0.549E6	20
	14%	0.647E6	23	0.315E6	11	0.671E6	25
	50%	0.890E6	32	1.174E6	43	0.790E6	29
25×50							
	3%	0.399E6	3	0.174E6	1	1.283E6	10
	8%	4.168E6	33	1.837E6	14	1.373E6	11
	14%	5.935E6	47	3.959E6	31	1.527E6	12
	50%	6.254E6	50	6.663E6	53	3.570E6	28
50×100							
	3%	36.304E6	36	5.529E6	5	6.008E6	6
	8%	41.699E6	41	37.227E6	37	7.253E6	7
	14%	47.781E6	47	57.622E6	57	10.946E6	10
	50%	65.174E6	65	76.325E6	76	12.338E6	12

Table 5.2 Separable Sparse Problems Results

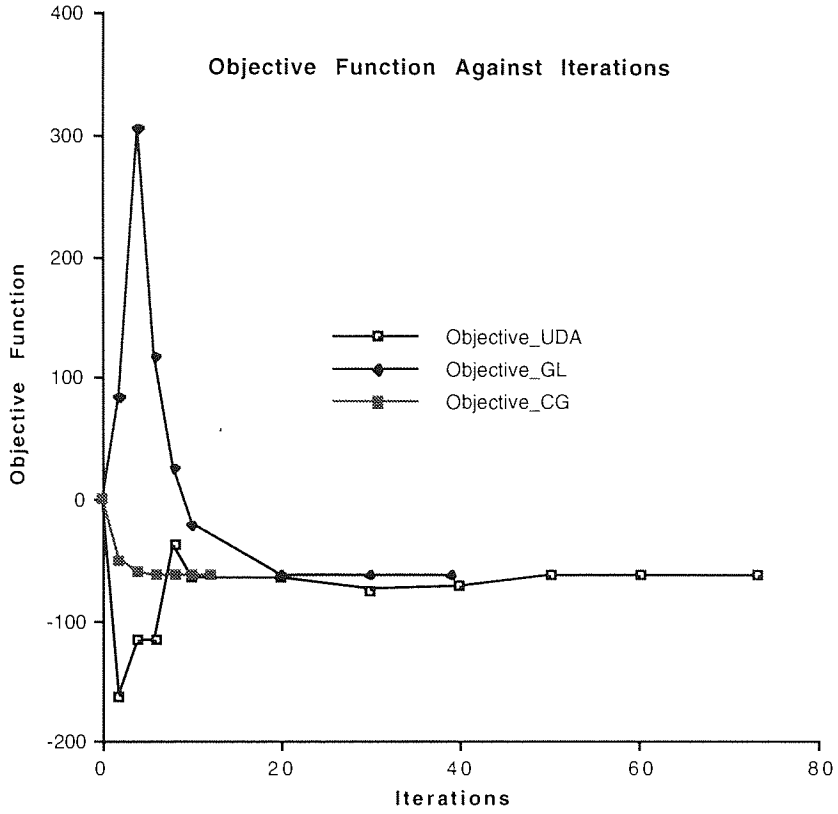

Figure 5.2 Change of the Objective Function Against the Number of Iterations

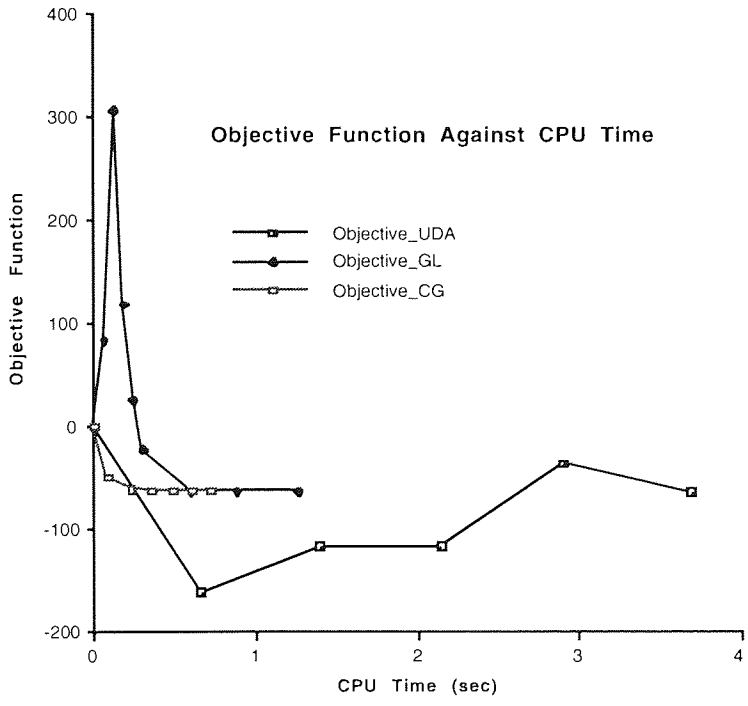

Figure 5.3 Change of the Objective Function Against CPU Time

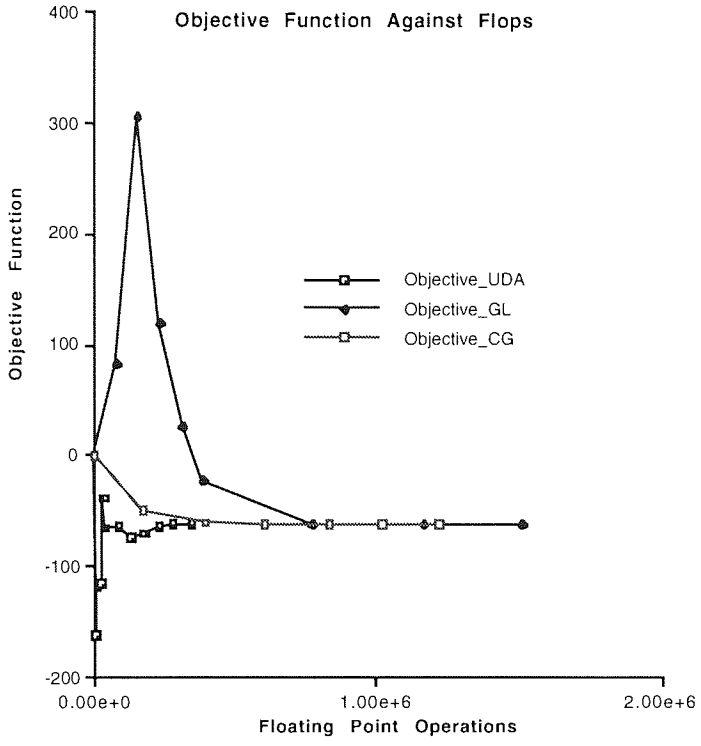

Figure 5.4 Change of the Objective Function Against Floating Point Operations

5.9.2 Tests on Random Generated Non-Separable QP Problems

The second set of experiments was considered non-separable convex quadratic program of the form **QP** with **Q** a symmetric matrix generated by the procedure SPRANDSYM of MATLAB and the RHS vector **b** generated in the same way as in the separable case. SPRANDSYM generates square positive definite matrices of a given size, density, and condition number. For the results recorded in the following tables the density was set to 60% and the condition number to 100. The termination criterion used is the same as in the last section. The results reported for the GL method pertain to the same values for parameters γ and σ as in the separable case. As in the last section, no initialization step was used.

No results are reported for the unified dual ascent algorithm in Tables 5.3, 5.4, 5.5, and 5.6 because of the way that method treats non-separable problems. As described in § 5.8.3, non-separable problems must be transformed into a separable form before they are solved by this method. That approach yields separable quadratic problems twice the size of the original non-separable one, so a fair comparison is not applicable. As mentioned in Lin and Pang, (1987) the equivalence between **QP** and **QPS** is valid even if $\delta = 0$. For $\delta = 0$, the objective in **QPS** is not strictly convex, however. This lack of strict convexity could invalidate the convergence results of the iterative method for solving **QPS**. For comparison reasons, the time, the number of flops and iterations required for the solution of small size quadratic problems by all three methods, including the unified dual ascent algorithm are reported in Table 5.7. Note that the results under the column UDA pertain to the performance of the algorithm applied directly to the non-separable problem without transformation ($\delta = 0$). The results recorded in Table 5.7 are for problems with five variables and three different densities of the **Q** matrix. The density of the constraint matrix was set to 14%. Table 5.7 shows that, even though convergence of the UDA algorithm can't be guaranteed for $\delta = 0$, this approach is very efficient for small size problems. However, numerical experience on larger problems indicates that convergence, if achieved, is slow and the method often fails to obtain an optimal solution.

From the results in Table 5.3, conclusions similar to the ones in the previous section can be drawn. The behaviour of the algorithms is the same except for the fact that now the conjugate gradient method is faster, even for the small problems with density of 50% or 100%.

From the experience gathered on these two sets of problems the following observations are apparent

- As expected, solving non-separable problems is more expensive than solving separable ones. The reason for this is that the matrix **Q** is denser compared to the separable case.
- There seems to be no relationship between the number of outer iterations in the conjugate gradient method and whether the problem solved is separable or not.

Size	CG		GL		RATIO
	Flops	Flops/n^3	Flops	Flops/n^3	GL/CG
5×10	0.152E6	152.736	0.217E6	217.569	1.427
10×20	1.092E6	136.550	2.559E6	319.926	2.345
15×30	4.434E6	164.224	10.926E6	404.696	2.463
25×50	26.676E6	213.415	67.577E6	540.619	2.535
50×100	313.116E6	313.116	809.903E6	809.903	2.584

Table 5.3 Non-Separable Dense Problems Results

- For inner iterations, however, one notices that non-separable problems are more demanding than separable ones. For separable problems, the number of iterations is usually much smaller than the number required for solving the corresponding non-separable problem.
- The GL method usually takes more iterations to converge than the conjugate gradient method and the number of iterations is proportionate to the size of the problem (the number of the variables).

As in the previous section, the results for the sparse non-separable problems are summarized in Tables 5.4, 5.5, and 5.6. They pertain to densities of 60%, 30%, and 12% of the **Q** matrix, respectively.

Studying the results in Tables 5.4 through 5.6 provides the following conclusions

- For problems where the **Q** matrix is sparse (12% density), GL_S outperforms CG_S for all densities of the constraint matrix **A**.
- For problems with denser **Q** matrices, GL_S is faster than CG_S only for small problems (n = 10).
- For bigger problems, CG_S performs better than GL_S. The savings in the number of floating point operations is around 50% except for problems with very sparse constraint matrices **A** (3% density), where it increases rather rapidly.

Size	Dens	CG_S Flops	CG_S Flops/n^3	GL_S Flops	GL_S Flops/n^3	RATIO GLS/CGS
5×10						
	3%	0.073E6	73.3	0.077E6	77.9	1.062
	8%	0.093E6	93.6	0.079E6	79.6	0.850
	14%	0.109E6	109.7	0.090E6	90.3	0.823
	50%	0.122E6	122.5	0.092E6	92.8	0.758
10×20						
	3%	0.416E6	52.1	0.790E6	98.8	1.898
	8%	0.425E6	53.1	0.900E6	112.5	2.304
	14%	0.893E6	111.7	0.904E6	113.0	1.011
	50%	1.226E6	153.3	1.076E6	134.6	0.877
15×30						
	3%	1.075E6	39.8	3.741E6	138.5	3.478
	8%	1.506E6	55.8	3.462E6	128.2	2.297
	14%	2.477E6	91.7	3.938E6	145.8	1.589
	50%	2.818E6	104.3	4.100E6	151,8	1.455
25×50						
	3%	5.549E6	44.4	21.170E6	169.3	3.814
	8%	12.679E6	101.4	21.415E6	171.3	1.688
	14%	13.997E6	111.9	20.916E6	167.3	1.494
	50%	30.032E6	240.2	28.172E6	225.3	0.938
50×100						
	3%	114.814E6	114.8	267.937E6	267.9	2.333
	8%	173.122E6	173.1	262.071E6	262.1	1.513
	14%	190.810E6	190.8	291.393E6	291.3	1.527
	50%	198.178E6	198.1	334.437E6	334.4	1.687

Table 5.4 Non-Separable Sparse Problems Results

Size	Dens	CG_S Flops	CG_S Flops/n^3	GL_S Flops	GL_S Flops/n^3	RATIO GLS/CGS
5×10						
	3%	80,905	80.9	52,382	52.3	0.647
	8%	89,164	89.1	44,645	44.6	0.500
	14%	108,487	108.4	53,299	53.2	0.491
	50%	112,797	112.7	55,032	55.03	0.487
10×20						
	3%	227,501	28.4	569,049	71.2	2.501
	8%	243,741	30.4	615,448	76.9	2.525
	14%	364,267	45.5	561,423	70.1	1.541
	50%	691,988	86.4	726,988	90.8	1.050
15×30						
	3%	765,407	28.3	2,166,345	80.2	2.830
	8%	1,088,520	40.3	2,103,402	77.9	1.932
	14%	1,606,771	59.5	1,904,938	70.5	1.185
	50%	2,538,863	94.03	2,867,476	106.2	1.129
25×50						
	3%	2,952,416	23.6	13,153,194	105.2	4.455
	8%	10,370,835	82.9	15,012,103	120.09	1.447
	14%	11,037,823	88.3	17,421,585	139.3	1.578
	50%	13,702,326	109.6	19,148,963	153.1	1.397
50×100						
	3%	73,060,309	73.06	193,354,875	193.3	2.646
	8%	120,566,441	120.5	201,928,879	201.9	1.674
	14%	135,059,858	135.05	209,699,900	209.6	1.552
	50%	148,270,614	148.2	226,883,290	226.8	1.530

Table 5.5 Non-Separable Sparse Problems Results

The convergence for a sparse non-separable problem with 20 variables is depicted in Figures 5.5 through 5.7.

Size	Dens	CG_S Flops	CG_S Flops/n^3	GL_S Flops	GL_S Flops/n^3	RATIO GLS/CGS
5×10						
	3%	33,947	33.9	18,282	18.2	0.538
	8%	36,726	36.7	18,710	18.7	0.509
	14%	60,349	60.3	15,210	15.2	0.252
	50%	170,677	170.6	37,894	37.8	0.222
10×20						
	3%	357,495	44.6	183,674	22.9	0.513
	8%	371,678	46.4	210,503	26.3	0.566
	14%	635,728	79.4	266,525	33.3	0.419
	50%	970,360	121.2	404,377	50.5	0.416
15×30						
	3%	628,520	23.2	670,530	24.8	1.066
	8%	703,689	26.06	700,112	25.9	0.994
	14%	1,975,105	73.1	1,169,405	43.3	0.592
	50%	2,396,663	88.7	1,718,459	63.6	0.717
25×50						
	3%	2,431,604	19.4	4,116,146	32.9	1.692
	8%	6,446,591	51.5	7,472,158	59.7	1.159
	14%	9,981,754	79.8	8,527,539	68.2	0.854
	50%	14,404,571	115.2	10,113,238	80.9	0.702
50×100						
	3%	69,345,961	69.3	133,191,153	133.1	1.920
	8%	136,135,270	136.1	134,465,393	134.4	0.987
	14%	138,701,812	138.7	148,830,916	148.8	1.073
	50%	189,749,949	189.7	169,956,673	169.9	0.895

Table 5.6 Non-Separable Sparse Problems Results

Studying the results of Table 5.7 one can see that the number of floating point operations required by the UDA algorithm is very small compared to those of the CG and the GL method. However, as far as CPU time is concerned, the UDA method has the worst performance among the tested algorithms.

	UDA			CG			GL		
Density	Time	Iter	Flops	Time	Iter	Flops	Time	Iter	Flops
60%	2.859	16	25,261	0.535	16	229,772	0.399	24	214,049
30%	2.742	16	22,849	0.359	11	153,454	0.389	24	213,774
12%	2.196	11	16,897	0.621	19	272,998	0.406	24	213,654

Table 5.7 Small Non-Separable Sparse Problems Results

5.9.3 Additional Runs

To test the performance of some variants of the first two described algorithms, some additional experiments were performed .

5.9.3.1 Conjugate Gradient Method

An inexact variant of the conjugate gradient method was run to see the effect of truncating on both the number of inner and outer iterations. This variant differs from the original conjugate gradient method in relation to the stopping criterion of both the inner and outer circle. The tolerance parameter ε in this variant was dynamically adjusted according to the value of the barrier parameter μ. The following expression for ε was used

$$\varepsilon^k = \max(\lambda\sqrt{\mu^k}, 10^{-7}),$$

where λ is a positive number smaller than 1. To prevent the tolerance parameter from becoming unreasonably small, ε was bound below by 10^{-7}. For $\lambda = 0$ one gets the original conjugate gradient method. For larger values of λ, a looser tolerance value is generated. The value of ε decreases as the method progresses, forcing greater accuracy. The method was

tested for different values of λ (0.001, 0.01, and 0.1). The reported results for the truncated variant pertain to $\lambda = 0.1$. The methods were run for n = 10, 20, 30, and 50 and m varied from 0.5n to n by a step-size of 2. Every problem was tested 10 times for both separable and non-separable problems. The value of the objective function was calculated for both variants and it was found that they have the same value at least until the second decimal digit.

From the above tests the following observations are noted

- For problems with $n - m = 1$, both methods are equal in regards to the number of inner and outer iterations.
- For problems with $m > \zeta n$, where $\zeta \in [0.6, 0.8]$, the truncated method converges in less iterations of the inner loop. As the size of the problem increases, ζ becomes bigger and the relative attractiveness of the truncated method tends to decrease rather rapidly.
- For problems with $m < \zeta n$, the truncated method performs worse than the exact variant.

5.9.3.2 Goldfarb and Lui Method

The GL method was run with three different pairs of values for the parameters γ and σ, both on separable and non-separable problems. The results can be found in Tables 5.8 and 5.9. Note that all variants were tested without the initialization step. The size and the density of the problems match those in Tables 5.1 and 5.3. For all the tested problems, the third variant stands out to be the clear winner. Taking into account the number of iterations, it becomes clear from these results that the third variant is more than 10 times faster than the first one and more than 70 times faster than the second. The cost per iteration was calculated in each case. As in all three variants, the cost is comparable and one can expect the same acceleration between the three variants for the running time and the total number of flops. The latter can be confirmed from the experimental results in Tables 5.8 and 5.9. All these conclusions are valid for both separable and non-separable problems. As in the previous section, the values of the objective function, corresponding to the solutions found by the three variants, agree until the second decimal digit, minimally.

Size		γ=0, δ=0.5, σ=0.166		γ=0.1, δ=0.1,σ=0.023		γ=0, δ=0.5, σ=1.5	
	Density	Iter	Flops(10E6)	Iter	Flops(10E6)	Iter	Flops(10E6)
5×10							
	100%	274	1.615	2026	11.956	24	0.141
10×20							
	100%	409	16.186	3002	118.805	39	1.543
15×30							
	100%	515	64.705	3775	474.295	51	6.407
25×50							
	100%	689	380.798	5032	2,781.1	70	38.687

Table 5.8 Results of Three Variants of the GL Method for Separable Problems

Size		γ=0, δ=0.5, σ=0.166		γ=0.1, δ=0.1,σ=0.023		γ=0, δ=0.5, σ=1.5	
	Density	Iter	Flops(10E6)	Iter	Flops(10E6)	Iter	Flops(10E6)
5×10							
	100%	274	2.487	2026	18.395	24	0.217
10×20							
	100%	409	26.857	2026	197.129	24	2.561
15×30							
	100%	515	110.374	3775	809.070	51	10.930
25×50							
	100%	689	665.223	5032	4,858.4	70	67.584

Table 5.9 Results of Three Variants of the GL Method for Non-Separable Problems

In Goldfarb and Lui, (1991), it was proven that parameters γ, δ, and σ are selected in a way that ensures the algorithm will drive the duality gap to zero by a fixed rate of $1 - \sigma/\sqrt{n}$, as the barrier parameter μ is reduced by the same multiple at each iteration. The value for σ used in the experiments performed for this research causes the barrier parameter and the value of the objective function to be reduced by a smaller rate. On the other hand, the number of iterations required for the algorithm to converge is reduced dramatically.

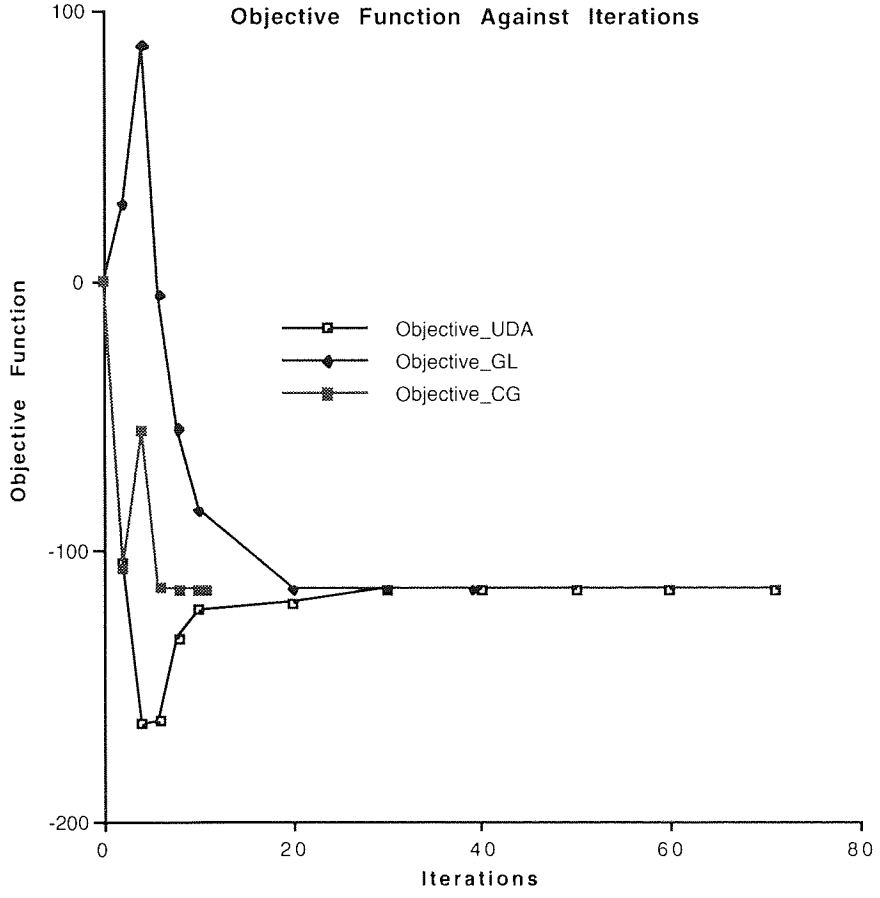

Figure 5.5 Change of the Objective Function Against the Number of Iterations

In Figures 5.3 and 5.6 the CPU time graphically represented is limited to the first 4 and 10 seconds respectively even though the convergence of the UDA method is not completed. This is done in view to show in detail the convergence process of the other two algorithms.

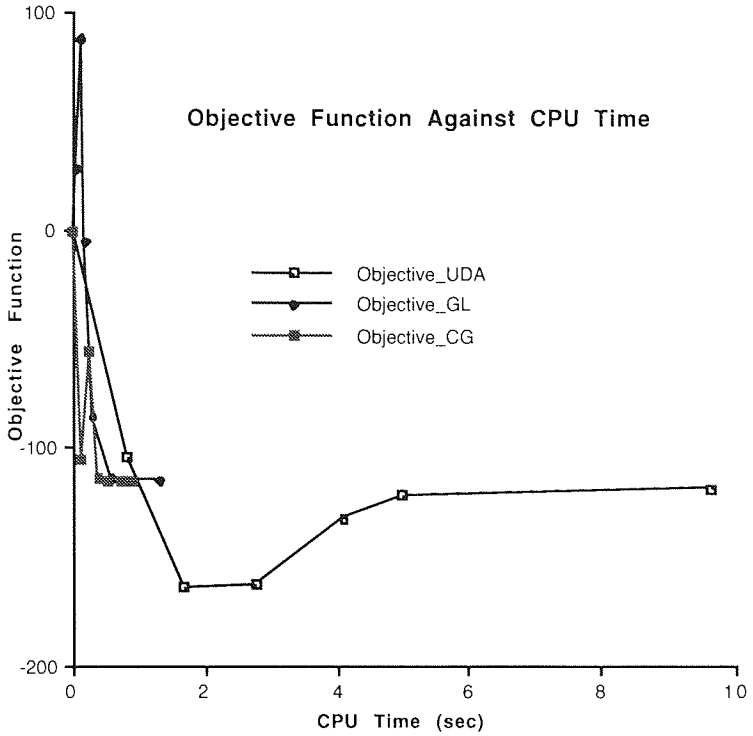

Figure 5.6 Change of the Objective Function Against CPU Time

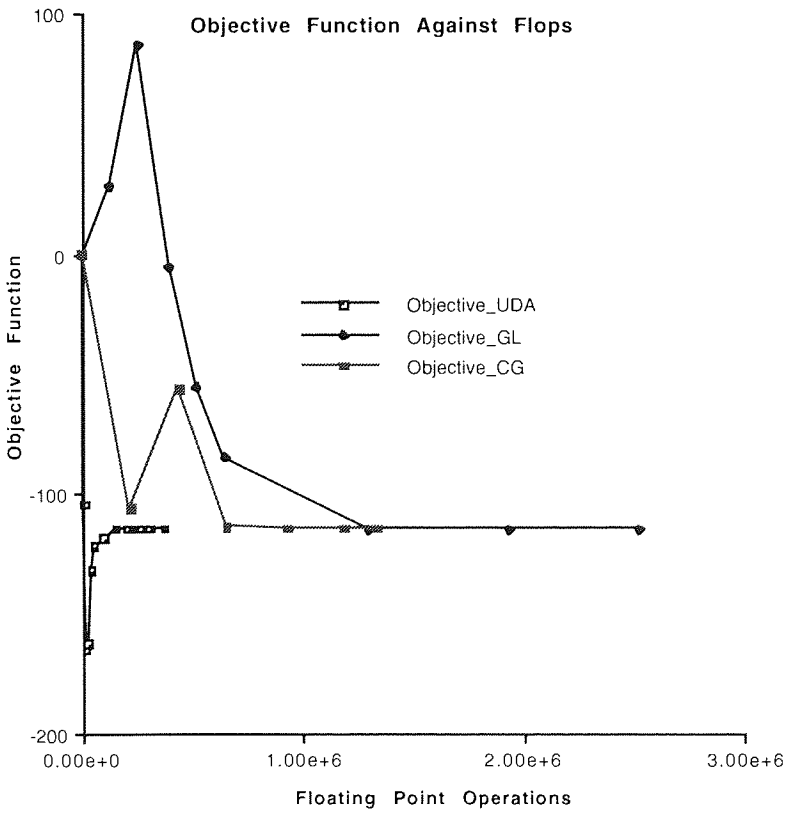

Figure 5.7 Change of the Objective Function Against Floating Point Operations

5.10 Predictor Corrector Method and Quadratic Programming

In the previous chapter the use of the predictor corrector method for solving linear programming problems was described. The present section is concerned with investigating the possible use of this technique as a centring scheme in the design of an algorithm for quadratic programming.

Consider the minimization quadratic programming problem in its standard form. The dual of this problem after introducing a vector of surplus variables is of the form:

$$\begin{aligned} &\max\ b^T y - \frac{1}{2} v^T Q v \\ &\text{s.t. } Qv + c - A^T y - z = 0 \\ &\qquad z \geq 0 \end{aligned}$$

The inequality constraint can be replaced by a logarithmic barrier term in the objective function and the problem is revised to

$$\begin{aligned} &\max\ b^T y - \frac{1}{2} v^T Q v + \mu \sum_{j=1}^{n} \ln(z_j) \\ &\text{s.t. } Qv + c - A^T y - z = 0. \end{aligned}$$

The corresponding Lagrangian to the above optimization problem is

$$L(x, v, y, z, \mu) = b^T y - \frac{1}{2} v^T Q v + \mu \sum_{j=1}^{n} \ln(z_j) - x^T (Qv + c - A^T y - z)$$

The first order conditions for the Lagrangian are

$$\begin{aligned} &\frac{\partial L(x,v,y,z,\mu)}{\partial z} \equiv XZe = \mu e, \\ &\frac{\partial L(x,v,y,z,\mu)}{\partial y} \equiv Ax = b, \\ &\frac{\partial L(x,v,y,z,\mu)}{\partial x} \equiv A^T y + z - Qv = c, \\ &\frac{\partial L(x,v,y,z,\mu)}{\partial v} \equiv Qv = x^T Q \end{aligned}$$

where $\mathbf{X}$, $\mathbf{Z}$ and $\mathbf{e}$ are defined as in the previous chapter. At the optimum stage, $\mathbf{v} = \mathbf{x}$. The last of the conditions is automatically satisfied at this point and $\mathbf{v}$ can be replaced with $\mathbf{x}$ in the third condition. Applying Mehrotra's predictor-corrector method to this set of conditions yields solving the affine system

$$\mathbf{X}\Delta\hat{\mathbf{z}} + \mathbf{Z}\Delta\hat{\mathbf{x}} = \mu\varepsilon - \mathbf{XZe}$$
$$\mathbf{A}\Delta\hat{\mathbf{x}} = \mathbf{b} - \mathbf{Ax}$$
$$\mathbf{A}^{\mathrm{T}}\Delta\hat{\mathbf{y}} - \Delta\hat{\mathbf{z}} + \mathbf{Q}\Delta\hat{\mathbf{x}} = \mathbf{z} + \mathbf{A}^{\mathrm{T}}\mathbf{y} - \mathbf{Qx} - \mathbf{c}$$

Note that, as in the linear case, the system is relaxed from cross products terms. The solution $\Delta\hat{\mathbf{x}}$, $\Delta\hat{\mathbf{z}}$ of this system can be used for the computation of primal and dual directions.

Based on the above procedure for generating both the primal and dual search directions one can outline an algorithm for quadratic programming problems.

Step 1. Find an initial feasible primal vector $\mathbf{x}$ to the problem.

Step 2. Generate $\mathbf{z}$ as an arbitrary vector.

Step 3. Compute a starting vector $\mathbf{y}$ satisfying $\mathbf{Qx} + \mathbf{c} - \mathbf{A}^{\mathrm{T}}\mathbf{y} - \mathbf{z} = \mathbf{0}$.

Step 4. Solve the modified system for $\Delta\hat{\mathbf{x}}$, $\Delta\hat{\mathbf{z}}$ and $\Delta\hat{\mathbf{y}}$.

Step 5. Use this solution to generate the search directions $\Delta\mathbf{x}$, $\Delta\mathbf{y}$, $\Delta\mathbf{z}$.

Step 6. Update vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ and the barrier parameter μ.

Step 7. Go back to step 4 until some convergence criterion is satisfied.

The main advantage of the described algorithm is that it reduces the quadratic problem to the solution of a system of so-called normal equations.

In Vanderbei, (1994) an implementation strategy based on solving the reduced Karush-Kuhn-Tucker (KKT) system is offered as an alternative to the predictor corrector method. For quadratic problems, the same derivation as in the linear case can be repeated except that the Hessian $\mathbf{Q}$ of the quadratic objective function is subtracted from $-\mathbf{ZX}^{-1}$ in the upper left block of the symmetric quasi-definite matrix. Assuming $\mathbf{Q}$ is positive semidefinite, it follows that this matrix is still quasi-definite and, so, any sparsity preserving the ordering scheme can be applied to solve the system. As this suggestion was not based on any practical results it is difficult to asses whether this would be worth further investigation.

5.11 Summary

In this chapter three different algorithms for convex quadratic programming were considered. Extensive tests on randomly generated separable and non-separable problems were carried out to study the effect of sparsity in both $\mathbf{Q}$ and the constraint matrix. The effect of the condition number of these matrices on the performance of the tested algorithms was also studied.

From the numerical results obtained, some conclusions about the performance of the truncated variant of the algorithm based on conjugate gradients were highlighted. Based on the experiments with the algorithm of Goldfarb and Lui, a new value for one of the parameters which dramatically improves the performance of the algorithm was also proposed.

The main purpose of this investigation of algorithms for quadratic programming is to determine whether they can be used efficiently in decomposition techniques for large-scale, sparse linear programming problems.

Chapter 6

Interior Point Algorithms and Decomposition of Linear Programs

6.1 Introduction

The development of efficient optimization techniques for large structured linear programs is of major significance in economic planning, engineering, and management science. An extensive literature exists on decomposition which shows the magnitude of the effort devoted to the subject (for an excellent review see Geoffrion, (1970)). Initially, the idea of decomposition, as suggested by Dantzig and Wolfe, (1960), was an extension of the use of the simplex method to solve large-scale and structured LP problems. With implementations of this idea, large LP problems arising in the oil industry, government, etc. were successfully solved. However, the decomposition algorithm, its variants, and many other methods based on different ideas never outclassed the simplex method in terms of labour involved (when these large problems can be handled by the simplex).

The relative "inefficiency" of decomposition algorithms developed in the previous decade may be due to their tight relationship with the simplex algorithm. It is, therefore, worthwhile to investigate decomposition in conjunction with interior point methods. By developing specialized solution algorithms using interior point methods to take advantage of the structure of the problems, significant gains in computational efficiency and reduction in computer memory requirements may be achieved. Such methods are mandatory for truly large problems, which can't otherwise be solved because of time and/or storage limitations. Decomposition is also attractive from the point of view of parallelism or concurrency.

Parallel processing is expected to speed up the solution process especially for problems with a large number of independent blocks.

Many new methods were developed, most of them based on penalty functions. Two algorithms following this principle will be described in this chapter. The first makes use of an augmented Lagrangian function, [Mulvey & Ruszcczynski, (1992)]. In the second, smoothed penalty functions are used to exploit the embedded network structure of the problem, [Zenios, et al., (1992)].

In addition, a new algorithm based on proximal point techniques appears promising for the solution of structured problems. The main advantage of this method is that the subproblems are totally decentralized, [Mahey & Tao, (1993)]. The term decentralized, discussed in § 6.4.2, means that there is no master problem involved in the solution process.

Finally, the classical idea of cutting planes, first proposed by Dantzig and Wolfe, is used in the theoretical paper of Goffin, et al., (1992). The new algorithm, which also makes use of an "oracle" to generate cutting planes, differs from the original in that instead of generating these planes at a point that optimizes the current LP relaxation of the associated MinMax problem, they are generated at "central points" of the so-called sets of localization. These sets of localizations are the polytopes given by the outer approximation of the epigraph associated with the current LP relaxation and limited above by the best objective value observed so far. This method is not implemented yet and no numerical results are reported.

In the following, the applicability of the interior point methods for linear and quadratic programming of Chapters 4 and 5 coupled with the latest developments in decomposition to structured LP problems is reviewed. The problems considered in this chapter are of block-angular structure with coupling constraints.

6.2 Structured LP Problems

Structure is an important attribute of large-scale linear programming. Large-scale programs almost always have distinctive structure besides convexity and linearity. The past

three decades have seen the identification of many classes of structured problems. There are many types of structure. However, the most common and most important are multi-divisional, combinatorial, dynamic, and stochastic structures. Of primal significance to this research are the multi-divisional problems which consist of interrelated subsystems to be optimized, [Geoffrion, (1970)]. The subsystems can be modules of an engineering system, reservoirs in a water resource system, departments or divisions of an organization, production units of an industry, or sectors of an economy. The interrelation between the subsystems is represented with the so called linking constraints or variables.

The general, block-diagonal, LP problem with linking constraints and linking variables is shown below in standard form.

$$\begin{aligned} &\text{Min } c_0^T y + \sum_{i=1}^{n} c_i^T x_i \\ &\text{s.t. } D_0 y + \sum_{i=1}^{n} B_i x_i = b_0 \\ &\quad D_i y + A_i x_i = b_i, \\ &\quad y,\ x_i \geq 0, \\ &\quad i=1,\ 2,\ \ldots,\ n. \end{aligned} \qquad (6.2.1)$$

Figure 6.1 gives a graphical representation of a block diagonal problem.

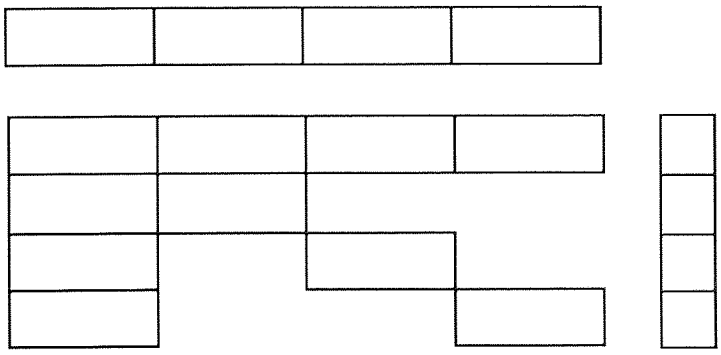

Figure 6.1 General Block Diagonal LP Problem

The most common structure in large LP problems is the block-angular structure, (see Figure. 6.2). In standard form, this structured LP problem is written as:

$$\text{Min} \sum_{i=1}^{n} c_i^T x_i$$

$$\text{s.t.} \sum_{i=1}^{n} B_i x_i = b_0 \qquad (6.2.2)$$
$$A_i x_i = b_i,$$
$$x_i \geq 0,$$
$$i = 1, 2, \ldots, n.$$

The dual to the above problem is

$$\text{Max } b_0^T y_0 + \sum_{i=1}^{n} b_i^T y_i$$

$$\text{s.t. } B_i^T y_0 + A_i^T y_i = c_i \qquad (6.2.3)$$
$$y_0, y_i \text{ unrestricted},$$
$$i = 1, 2, \ldots, n.$$

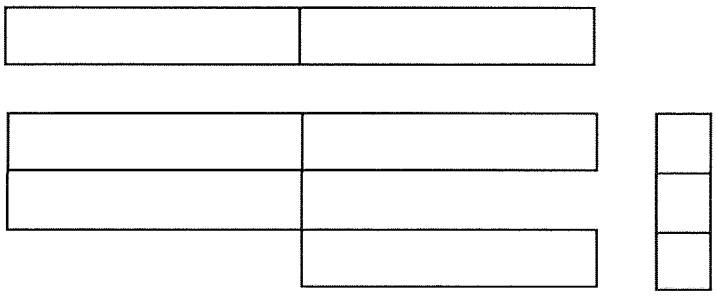

Figure 6.2 Diagram of a 2-Block LP Problem

Block-angular structured problems with coupling variables are amenable to block-angular problems with coupling constraints and, thus, can be solved by forming their dual first.

6.3 Advantages of Decomposition

Areas of application exist which require the solution of optimization models with tens of thousands of variables. Even with recent developments in mathematical programming, motivated by Karmarkar's interior point method, [Karmarkar, (1984); Marsten, et al.,

(1990)], or with advances in super-computing technology, [Meyer & Zenios, (1988)], such large-scale problems defy solution with general purpose software. It is, therefore, essential to design algorithms that exploit the structure of the problem. When the underlying structure is pervasive in several applications, such an approach is not only very effective, but is, also, well justified, [Zenios, et al., (1990)].

The main purpose of decomposition is to exploit the inherent parallelism of block-angular structured problems, using the relative independence of the subproblems.

It is assumed that solving a set of linear subproblems is easy, relative to the entire original problem. This is quite reasonable for most applications, since the subproblems are solved independently, and may, therefore, be solved in parallel. This is particularly appropriate in a MIMD computational environment. Secondly, some types of structure in the subproblems may be exploited which could not be directly used if the original problem were solved with the entire set of constraints. An example of this is the multicommodity flow problem, where each constraint submatrix is a node-arc-incidence matrix. In this case, each subproblem may be solved with a special-purpose-network code. And third, note that the difficulty of solving most linear programs (in some sense the easiest of optimization problems) in practice increases as a cubic function in proportion to the size of the problem. As such it is much more efficient to solve a set of small problems than a single aggregated problem. Based on this logic, the basic effort in the decomposition of structured LP problems has been to devise solution methods that break the problems down into a sequence of subproblems in lower-dimensional spaces, [Cohen, (1980); Dantzig & Wolfe, (1960); Geoffrion, (1970); Ladson, (1970); Mulvey & Ruszczynski, (1992); Mahey & Tao, (1993); Pinar & Zenios , (1992)].

The following outlines a model algorithm for decomposing structured linear problems.

Step 1. Eliminate the coupling constraints so the resulting equivalent problem has a decomposable constraint matrix. Set k = 0 and repeat the following steps.

Step 2. If the objective function is decomposable, let $\mathbf{x}_k^i$ be the current estimate of the optimal vector $\mathbf{x}^*$ corresponding to the i-th independent subproblem. If the objective is not

decomposable, then special algorithms must be used. The purpose of these algorithms is to convert the objective to a decomposable form or at least to approximate it as such.

Step 3. Test convergence.

If the convergence conditions are satisfied at $\mathbf{x_k}$, the algorithm terminates with $\mathbf{x_k}$ as the solution.

Step 4. Compute a feasible search direction.

Compute a non-zero vector $\mathbf{p_k}$, the direction of search.

Step 5. Compute a step-length.

Compute a positive scalar α_k, the step-length, for which holds $F(\mathbf{x_k}+\alpha_k\mathbf{p_k}) < F(\mathbf{x_k})$.

Step 6. Update the estimate of the minimum.

Set $\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha_k\mathbf{p_k}$, $k = k + 1$ and go to Step 1.

6.4 New Approaches to Decomposition

In this section three different, recent approaches to the decomposition of structured LP problems will be described. All algorithms handle problems in standard form, i.e., all constraints are equalities. For the first of the algorithms described it is assumed that the optimal solution is in the positive orthant but there are no upper bounds on the variables. The rest of the algorithms mentioned in this chapter handle lower and upper bounds on the variables explicitly. In what follows it is assumed that the decomposition problems are feasible, i.e. the solution set is not empty.

All methods make use of penalty functions to eliminate the linking constraints. These functions are based on the violation of the coupling constraints.

The main difference among the algorithms is in the way they introduce separability in the objective function after the elimination of the coupling constraints. In the algorithm of Mulvey and Ruszczynski that is achieved by quadratic approximation. Zenios and Pinar use simplicial decomposition. Finally, the algorithm of Mahey and Tao yields totally decentralized subproblems using proximal point techniques.

6.4.1 A Decomposition Method Based on Augmented Lagrangian

This decomposition approach, as described in Mulvey and Ruszczynski, (1992), is based on the use of an augmented Lagrangian method to handle the coupling constraints. Using a separable diagonal quadratic approximation for the augmented Lagrangian, independent quadratic subproblems are obtained. These problems can be readily solved in parallel by any of the quadratic solvers of Chapter 5.

Another method based on augmented Lagrangians has been suggested by Ruszczynski, (1989). It has linear subproblems identical with the Dantzig-Wolfe method, but its master problem is quadratic with only n general upper bound-type constraints, [Mulvey & Ruszcczynski, (1992)].

It could seem strange to solve a linear program by solving a sequence of quadratic programs because quadratic problems are, in general, more difficult to solve. However, for decomposable structure problems as those studied in this chapter, the use of quadratic programming is justified because the size of the quadratic problems generated is equal to the size of the independent subproblems or the quadratic master problem is of small size and of special form, which is easily solved.

A step of the augmented Lagrangian method can be stated for the solution of (6.2.2) as follows

Algorithm 6.1 <u>**Outer Loop**</u> (The Augmented Lagrangian Method)

Step 1. For fixed multipliers π^k solve the problem

$$\begin{aligned} &\textbf{Min } L_r(x, \pi^k) \\ &\textbf{s.t. } x_i \in X_i, \\ &\qquad i = 1, 2, \ldots, n, \end{aligned} \qquad (6.4.1)$$

where

$$L_r(x, \pi) = \sum_{i=1}^{n} c_i^T x_i + \pi^T \left(b_0 - \sum_{i=1}^{n} B_i x_i \right) + \frac{1}{2} r \left| b_0 - \sum_{i=1}^{n} B_i x_i \right|^2 \qquad (6.4.2)$$

$r > 0$ is a penalty parameter and $\mathbf{X_i}$ is the feasible set for subvector $\mathbf{x_i}$ of vector $\mathbf{x}$

$$\mathbf{X_i} = \{\mathbf{x_i} \in R^{n_i} : \mathbf{A_i x_i} = \mathbf{b_i},\ \mathbf{x_i} \geq \mathbf{0}\},\ i = 1, 2, \ldots, n.$$

Let $\mathbf{x}^k = (x_1^k, x_2^k, \ldots, x_n^k)$ be the solution of (6.4.1)

Step 2. If $\sum_{i=1}^{n} \mathbf{B_i x_i} = \mathbf{b_0}$ then stop (optimal solution found); otherwise, set

$$\pi^{k+1} = \pi^k + r\left(\mathbf{b}_0 - \sum_{i=1}^{n} \mathbf{B}_i x^k{}_i\right),$$

increase k by 1 and go to Step 1.

The main disadvantage of the algorithm as described above is that the objective function $\mathbf{L}_r(\mathbf{x}, \pi)$ of the quadratic problem (6.4.1) is non-separable, and so no decomposition is obtained. The problem of separability of that function will be considered later in this section.

For the sake of completeness the theoretical justification of the above algorithm is presented, [Bertsekas, (1982)]. Let us denote by $f(\mathbf{x})$, $h(\mathbf{x}) = \mathbf{0}$ and $\mathbf{X}$ the objective function, the coupling constraints, and the decomposable block of constraints of problem (6.2.2) respectively.

Proposition: We can assume, without loss of generality that $f(\mathbf{x})$ and $h(\mathbf{x})$ are continuous functions and that the set $\mathbf{X}$ is closed. If for $k = 0, 1, \ldots$ vector $\mathbf{x_k}$ is a global minimum of problem (6.4.1) where $\{\pi_k\}$ is bounded and $0 < r_k < r_{k+1}$ for all k, $r_k \to \infty$, then every limit point of the sequence $\{\mathbf{x_k}\}$ is a global minimum of problem (6.2.2)

Proof: Let $\mathbf{x}^*$ be a limit point of $\{\mathbf{x_k}\}$. By definition of $\mathbf{x_k}$

$$\mathbf{L}_{rk}(\mathbf{x_k}, \pi_k) \leq \mathbf{L}_{rk}(\mathbf{x}, \pi_k)\ \forall\ \mathbf{x} \in \mathbf{X} \qquad (\mathbf{6.4.3})$$

Let f^* denote the optimal value of the original problem. We have

$$f^* = \inf_{\mathbf{x} \in \mathbf{X}, h(\mathbf{x})=0} f(\mathbf{x}) = \inf_{\mathbf{x} \in \mathbf{X}, h(\mathbf{x})=0} \mathbf{L}_{rk}(\mathbf{x}, \pi_k).$$

Hence, by taking the infimum of the right-hand side of (6.4.3) over $\mathbf{x} \in \mathbf{X}$, $h(\mathbf{x}) = \mathbf{0}$, the following is obtained

$$L_{rk}(\mathbf{x_k}, \pi_\mathbf{k}) = f(\mathbf{x_k}) + \pi_\mathbf{k}'h(\mathbf{x_k}) + \frac{1}{2} r_k|h(\mathbf{x_k})|^2 \leq f^*.$$

The sequence $\{\pi_\mathbf{k}\}$ is bounded and hence it has a limit point π^*. Without loss of generality, it may be assumed $\pi_\mathbf{k} \rightarrow \pi^*$. By taking the limit superior in the relation above and by using the continuity of functions $f(\mathbf{x})$ and the $h(\mathbf{x})$, the following is obtained

$$f(\mathbf{x}^*) + \pi^{*\prime}h(\mathbf{x}^*) + \lim_{k\rightarrow\infty}\sup \frac{1}{2} r_k\, |h(\mathbf{x_k})|^2 \leq f^* \qquad (\mathbf{6.4.4})$$

Since $|h(\mathbf{x_k})|^2 \geq \mathbf{0}$, $r_k \rightarrow \infty$, it follows that $h(\mathbf{x_k}) \rightarrow \mathbf{0}$ and

$$h(\mathbf{x}^*) = \mathbf{0}, \qquad (\mathbf{6.4.5})$$

for otherwise the limit superior in the left-hand side of (6.4.4) will equal $+\infty$. Since $\mathbf{X}$ is a closed set it is also obtained that $\mathbf{x}^* \in \mathbf{X}$. Hence $\mathbf{x}^*$ is feasible, and

$$f^* \leq f(\mathbf{x}^*). \qquad (\mathbf{6.4.6})$$

Using (6.4.4), (6.4.5), and (6.4.6), the resulting formula is

$$f^* + \lim_{k\rightarrow\infty}\sup \frac{1}{2} r_k\, |h(\mathbf{x_k})|^2 \leq f(\mathbf{x}^*) + \lim_{k\rightarrow\infty}\sup \frac{1}{2} r_k\, |h(\mathbf{x_k})|^2 \leq f^*.$$

Hence,

$$\lim_{k\rightarrow\infty} \frac{1}{2} r_k\, |h(\mathbf{x_k})|^2 = \mathbf{0} \text{ and } f(\mathbf{x}^*) = f^*,$$

which proves that $\mathbf{x}^*$ is a global minimum for problem (6.2.2). Q.E.D.

The termination criterion used in Step 2 is justified by the fact that in Step 1 of the algorithm the independent subproblems are optimized over the corresponding part of the coupling constraints, so if the solution found by Step 1 satisfies the whole set of coupling constraints then this solution must be the optimal.

There are several advantages to the augmented Lagrangian approach over the usual dual methods. Simplicity and stability of multiplier iterations and the possibility of starting from arbitrary π^0 are among the most important ones. The well-known Dantzig-Wolfe decomposition method may be viewed as a dual method based on the Lagrangian function

$$\Lambda(\mathbf{x}, \pi) = \sum_{i=1}^{n} \mathbf{c}_i^T \mathbf{x}_i + \pi^T \left(\mathbf{b}_0 - \sum_{i=1}^{n} \mathbf{B}_i \mathbf{x}_i \right).$$

The above function is separable into terms dependent on $\mathbf{x}_i$, i = 1, 2, ...,n and can be minimized over $\mathbf{x}_i$ in the decomposable blocks of the structured LP independently for each i. However, updating π requires the solution of a linear master problem which has the number or rows equal to the number of the coupling constraints and unspecified number of columns.

It is known that if (6.2.2) has a solution, then the augmented Lagrangian algorithm is finitely convergent. However, (6.4.2) is non-separable, so problem (6.4.1) cannot be split into n independent subproblems. There are several possible ways to overcome this difficulty, including the use of alternating direction methods, [Fortin & Glowinski, (1983)], which results in the easily parallelized Progressive Hedging Algorithm and the Simplicial Decomposition Algorithm described later in this chapter.

Alternating direction methods introduce additional variables $\xi = (\xi_1, \xi_2, ..., \xi_n)$ to the problem to replace the coupling constraints by a new set of conditions:

$$\mathbf{x}_i - \xi_i = \mathbf{0}, \; i = 1, 2, ..., n$$

$$\sum_{i=1}^{n} \mathbf{B}_i \xi_i = \mathbf{b}_0$$

If one writes down the augmented Lagrangian terms corresponding to the terms $\mathbf{x}_i - \xi_i = \mathbf{0}$ (with multipliers $\mathbf{u}$), it shall be seen that in problem (6.2.2) minimization with respect to $\mathbf{x}$, after ignoring constant terms, is decomposable (for fixed ξ) into n subproblems

$$\textbf{Min } \mathbf{c}_i^T \mathbf{x}_i - \mathbf{u}_i^T \mathbf{x}_i + \frac{1}{2} r \, \|\mathbf{x}_i - \xi_i\|^2$$

$$\textbf{s.t. } \mathbf{x}_i \in \mathbf{X}_i.$$

Minimization in ξ (for fixed $\mathbf{x}$) is obvious because the corresponding problem

$$\text{Min } -u\xi - u(\sum_{i=1}^{n} B_i\xi_i - b_0) + \frac{1}{2} r \sum_{i=1}^{n} \|x_i - \xi_i\|^2 + \frac{1}{2} r \sum_{i=1}^{n} \|B_i\xi_i - b_0\|^2)$$

is unconstrained. Note that the solution obtained for ξ will now satisfy the coupling constraints. It is, therefore, possible to derive a block-wise Gauss-Seidel method for solving problem (6.2.2) with alternating steps made in $\mathbf{x}$ and ξ. Multiplier updates can be made after each alternating direction iteration. Numerical experience, [Mulvey & Vladimirou, (1991); Vladimirou, (1990)], indicates that in some cases convergence of the method can be slow.

As an alternative to that method for this algorithm, the augmented Lagrangian function is locally approximated by a separable linear-quadratic function. Combining the updates of the separable approximation with the quadratic solvers of Chapter 5 gives a decomposition algorithm for large-scale, LP problems.

In the following a description of the separable approximation of the augmented Lagrangian will be presented as well as the final algorithm and the nature of the resulting quadratic subproblems.

It is clear that non-separability of (6.4.2) is due to the quadratic term

$$\Phi_r(\mathbf{x}) = \frac{1}{2} r \left| b_0 - \sum_{i=1}^{n} B_i x_i \right|^2 , \qquad (6.4.7)$$

which contains cross-products

$$\phi_{ij}(x_i, x_j) = < B_i x_i, B_j x_j >, \; i \neq j. \qquad (6.4.8)$$

However, supposing that $\mathbf{x}$ belongs to a neighbourhood of some reference point $\mathbf{x}^*$, (6.4.8) is approximated locally by

$$\phi^*_{ij}(x_i, x_j; x^*) = <B_i x_i, B_j x_j^*> + <B_i x_i^*, B_j x_j> - <B_i x_i^*, B_j x_j^*> \qquad (6.4.9)$$

with an error of order $O(\| \mathbf{x} - \mathbf{x}^* \|^2)$. Using the approximation (6.4.9) instead of the quadratic term (6.4.8) in (6.4.7), after a simple transformation process, the following is obtained

$$\Phi_r(x) \equiv -r\,\frac{n-1}{2}\,|b_0|^2 + \sum_{i=1}^{n}\left\{\frac{1}{2}\,r\left|b_0 - B_i x_i - \sum_{j\neq i} B_j x_j^*\right|^2 + r\left\langle b_0 - \frac{1}{2}\sum_{j=1}^{n} B_j x_j^*,\ \sum_{j\neq i} B_j x_j^*\right\rangle\right\}.$$

Thus, in the neighbourhood of x^*, the problem (6.4.1) can be approximated, after ignoring constant terms, by n subproblems

$$\text{Min } L^{*i}{}_r(x_i, \pi; x^*) = c_i^T x_i - \pi^T B_i x_i + \frac{1}{2}\,r\left|b_0 - B_i x_i - \sum_{j\neq i} B_j x_j^*\right|^2$$

$$\text{s.t. } A_i x_i = b_i \qquad (6.4.10)$$
$$x_i \geq 0$$
$$i = 1, 2, \ldots, n.$$

Using the above transformation one can describe an inner iteration of the augmented Lagrangian algorithm for solving (6.4.1) as follows.

Algorithm 6.1 Inner Loop

Step 0. Set (a) $\pi = \pi^k$, (b) $x^{*k,m} = x^{k-1}$, and (c) $m = 1$, where m is the counter of the iterations in the inner loop

Step 1. For $i = 1, 2, \ldots, n$, solve the quadratic programming problem (6.4.10) with $x^* = x^{*k,m}$ obtaining new points $x_i^{k,m}$.

Step 2. If $|B_i(x_i^{k,m} - x_i^{*k,m})| \leq \varepsilon$, $i = 1, 2, \ldots, n$, where $\varepsilon > 0$ is some prescribed accuracy, then Stop;

otherwise, set

$$x^{*k+1,m} = x^{*k,m} + \tau(x^{k,m} - x^{*k,m}),\ \tau \in (0,\ 1),$$

increase m by 1 and go to Step 1.

The quadratic subproblems (6.4.10) are convex with

$$Q = rB_i^T B_i \text{ and } c = c_i - B_i^T\pi - rB_i^T\left(b_0 - \sum_{j\neq i} B_j x^*{}_j\right),$$

so any of the algorithms described in Chapter 5 can be used for their solution.

6.4.2 A Decomposition Algorithm Based on Proximal Point Techniques

A particularly interesting algorithm is considered in this section. The novel element in this technique is that the subproblems are totally decentralized. That is something which is, in general, not possible with the classical methods for decomposition. To achieve this, proximal point methods are used.

The Proximal Point Algorithm is based on the Moreau-Yosida regularization of a maximal monotone operator and has been analysed by Rockafellar, (1976) in the context of convex analysis. Proximal methods of multipliers combined with a two-metric gradient-projection approach were successfully used for the solution of very large-scale linear programming programs, [Wright, (1990)].

The decomposition approach described in this section, [Mahey & Tao, (1993)], is based on a scaled and relaxed version of Spingarn's Partial Inverse method to minimize a convex separable function on a subspace. The Partial Inverse method, which is a certain constrained version of the Proximal Point algorithm, uses a transformation of the problem to build a coupling subspace with a very simple structure.

The algorithm starts by first building copies of the multipliers $\mathbf{u}$ associated with the coupling constraints and defining the problem

$$\begin{aligned} &\textbf{Max} \quad H(U) = \sum_{i=1}^{n} h_i(u_i) \\ &\textbf{s.t.} \quad U \in E, \end{aligned}$$

where $h_i(u_i) = \{\min x_i(c_i + u_i B_i),\ \text{s.t. } A_i x_i = b_i\}$ and E is the coupling subspace.

Algorithm 6.2 (The Decentralized Algorithm)

Step 0. (a) Initialize the coupling subspace

$$E \equiv \left\{ V = (v_1, v_2, \ldots, v_n) \mid \sum_{i=1}^{n} v_i = 0 \right\}$$

and (b) let **B** be the orthogonal subspace to **E**, i.e.

$$\mathbf{B} \equiv \{\mathbf{U} = (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n) \mid \mathbf{u}_1 = \mathbf{u}_2 \ldots = \mathbf{u}_n\}.$$

The elements of each vector $\mathbf{u}_i$ are randomly generated numbers. Start with arbitrary $\mathbf{x}^0$ and (c) set k = 1.

Step 1. Let $\mathbf{x}_i^{k+1}$ be the optimal solution of the quadratic subproblems

$$\begin{aligned} &\textbf{Min } (\mathbf{c}_i + \mathbf{u}_i^k \mathbf{B}_i)\mathbf{x}_i + \frac{\lambda}{2} \left\| \mathbf{B}_i \mathbf{x}_i + \mathbf{v}_i^k - \mathbf{b}_{0i} \right\|^2 \\ &\textbf{s.t. } \mathbf{A}_i \mathbf{x}_i = \mathbf{b}_i, \\ &\quad \mathbf{x}_i \geq \mathbf{0}, \\ &\quad i = 1, 2, \ldots, n, \end{aligned} \qquad (6.4.11)$$

where the scaling parameter $\lambda > 0$, and $\mathbf{b}_{0i}$ is the portion of $\mathbf{b}_0$ corresponding to the partition i of the coupling constraints. If the solution $\mathbf{x}^k = [\mathbf{x}_1^k, \mathbf{x}_2^k, \ldots, \mathbf{x}_n^k]$ satisfies some optimality criterion, then stop. Otherwise, perform Steps 2 and 3 and proceed from Step 1.

Step 2. Compute $\mathbf{u}'_i$ and $\mathbf{v}'_i$

$$\begin{aligned} &\text{(a) } \mathbf{u}'_i = \mathbf{u}_i^k + \lambda(\mathbf{B}_i \mathbf{x}_i^{k+1} + \mathbf{v}_i^k - \mathbf{b}_{oi}), \\ &\text{(b) } \mathbf{v}'_i = \mathbf{b}_{oi} - \mathbf{B}_i \mathbf{x}_i^{k+1}. \end{aligned}$$

Step 3. Projection step.

$$\begin{aligned} &\text{(a) } \mathbf{U}^{k+1} = \mathbf{Proj}_{\mathbf{E}}(\mathbf{u}'_1, \mathbf{u}'_2, \ldots, \mathbf{u}'_n), \\ &\text{(b) } \mathbf{V}^{k+1} = \mathbf{Proj}_{\mathbf{B}}(\mathbf{v}'_1, \mathbf{v}'_2, \ldots, \mathbf{v}'_n), \end{aligned}$$

where $\mathbf{u}_{ij}^{k+1} = \mathbf{Proj}_{\mathbf{E}}\mathbf{u}'_{ij}$ and $\mathbf{v}_{ij}^{k+1} = \mathbf{Proj}_{\mathbf{B}}\mathbf{v}'_{ij}$ is respectively equivalent to

$$\mathbf{u}_{ij}^{k+1} = \mathbf{u}'_{ij} - \frac{1}{n}\sum_{i=1}^{n}\mathbf{u}'_{ik} \text{ and } \mathbf{v}_{ij}^{k+1} = \mathbf{v}'_{ij} + \frac{1}{n}\sum_{i=1}^{n}\mathbf{u}'_{ik}.$$

The use of vector **v** in algorithm 6.2 is reminiscent of the resource-directive approach, [Geoffrion, (1970)]. Vector **u** can be seen as the price vector in a price-directive approach applied to the decentralized system whose subproblems are of the form (6.4.11). Algorithm 6.2 iteratively minimizes a separable function depending both on prices (**u**) and resources (**v**). In convex programs the prices converge to a Karush-Kuhn-Tucker vector and

the resources to an optimal allocation of the original problem, [Spingarn, (1985)]. The algorithm is in this sense both price- and resource-directive, though it is not primal feasible.

In Mahey and Tao's paper the projection used in Step 3 is left unclear. However, to clarify this, the projections used in algorithm 6.2 are based on the condition that **E** and **B** must be orthogonal. Let **H** denote the space of all pairs $(\mathbf{x}, \mathbf{v})$, where $\mathbf{x} \in R^n$ and $\mathbf{v}$ is a $m \times n$ array of real numbers. If R^n is given the inner product

$$\langle \mathbf{x}, \mathbf{x}' \rangle = \frac{1}{n}\langle x_1, x_1' \rangle + \frac{1}{n}\langle x_2, x_2' \rangle + \ldots + \frac{1}{n}\langle x_n, x_n' \rangle,$$

where $\langle x_j, x_j' \rangle$ is a arbitrary inner product on R^n, the subspaces **E** and **B** are orthogonal, if **H** is endowed with the inner product

$$\langle (\mathbf{x}, \mathbf{v}), (\mathbf{x}', \mathbf{v}') \rangle = \langle \mathbf{x}, \mathbf{x}' \rangle + \frac{1}{n}\sum_{ij} v_{ij} v_{ij}'.$$

The choice of the inner product $\langle (\mathbf{x},\mathbf{v}),(\mathbf{x}',\mathbf{v}') \rangle$ is made somewhat arbitrarily, [Spingarn, (1985)]. If p_{ij} are arbitrary positive numbers, it is equally natural to use any inner product satisfying

$$\langle (\mathbf{x}, \mathbf{v}), (\mathbf{x}', \mathbf{v}') \rangle = \sum_{j}(x_j, x_j') + \sum_{ij} p_{ij} v_{ij} v_{ij}'.$$

Associated to this inner product is the subspace $\mathbf{E}' \equiv \{ \mathbf{V} = (v_1, \ldots v_n) \mid \sum_{ij} p_{ij} v_{ij} = 0 \}$. Letting $p_i = \sum_k p_{ik}$ the projection of vectors **v** and **u** into **B** and **E'** are given by the formulae

$$v_{ij}^{k+1} = v_{ij}^{k} + \sum_{k} p_{ik} u_{ik}^{k} / p_i \text{ and } u_{ij}^{k+1} = u_{ij}^{k} - \sum_{k} p_{ik} u_{ik}^{k} / p_i.$$

In the implementation used for the tests performed in this chapter the values of parameters p_{ik} were set to one, for all $k = 0, 1, \ldots, n$. That results in $p_i = \sum_k p_{ik} = n$.

The quadratic problems generated in Step 1 of 6.2 are convex because they have $\mathbf{Q} = \frac{\lambda}{2}\mathbf{B}_i^T \mathbf{B}_i$, which is a positive definite square matrix. For the solution of these problems any of the algorithms for convex quadratic programming described in the previous chapter could be used.

6.4.3 A Decomposition Approach Based on Smoothed Exact-Penalty Functions

The idea of using penalty function methods to simplify optimization problems is probably as old as the field of non-linear programming itself. The method described in this section was proposed by Zenios, et al., (1990) and is based on smoothed, exact-penalty functions. These functions combine the best of exact-penalty functions and augmented Lagrangians and appear to be well suited to large-scale programming.

The algorithm was first proposed to solve non-linear problems of the form

$$\begin{aligned} \textbf{Min} \quad & \mathbf{f(x)} \\ \textbf{s.t.} \quad & \sum_{i=1}^{n} \mathbf{B_i x_i} = \mathbf{b_0} \\ & \mathbf{Ex} \leq \mathbf{b}, \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned} \qquad (6.4.12)$$

but can handle linear problems of the form (6.2.2) by modifying the penalty function as described later in this section. The use of smoothed, exact penalty to eliminate side constraints motivates the following algorithmic framework.

Algorithm 6.3 (The Linear-Quadratic Smoothed-Penalty Algorithm)

Step 0. (a) Find an initial feasible solution for the relaxed problem

$$\begin{aligned} \textbf{Min} \quad & \sum_{i=1}^{n} \mathbf{c_i^T x_i} \\ \textbf{s.t.} \quad & \mathbf{A_i x_i} = \mathbf{b_i}, \\ & \mathbf{x_i} \geq \mathbf{0}, \\ & i = 1, 2, \ldots, n. \end{aligned} \qquad (6.4.13)$$

(b) Set k = 0 and (c) let $\mathbf{x^0}$ be the optimal solution of this problem. (d) If $\mathbf{x^0}$ satisfies the coupling constraints within a prescribed error ε^{opt} , then stop. Otherwise, choose $\mu^0 > 0$, $\varepsilon^0 > 0$ and go to Step 1.

Step 1. (a) Calculate the violation $\mathbf{t_j} = \left(\sum_{i=1}^{n} \mathbf{B_i x_i} - \mathbf{b_0} \right)_j$ for all j. Using t_j as a starting point for evaluating the penalty function ϕ,

$$\text{where } \phi = \begin{cases} 0 & \text{if } \mathbf{t} \le 0, \\ \dfrac{\mathbf{t}^2}{2\varepsilon} & \text{if } 0 \le \mathbf{t} \le \varepsilon, \\ \mathbf{t} - \dfrac{\varepsilon}{2} & \text{if } \mathbf{t} \ge \varepsilon, \end{cases}$$

(b) solve the problem

$$\begin{aligned} \textbf{Min } & \Phi_{(\mu,\varepsilon)}(x) = \sum_{i=1}^{n} \mathbf{c}_i^T \mathbf{x}_i + \mu \sum_{j=1}^{s} \phi(\varepsilon, \mathbf{t}_j) \\ \textbf{s.t. } & \mathbf{diag}[\mathbf{A}_1\, \mathbf{A}_2 \ldots \mathbf{A}_n]\mathbf{x} = \mathbf{b}, \\ & \mathbf{x} \ge \mathbf{0}, \\ & i = 1, 2, \ldots, n, \end{aligned} \qquad (6.4.14)$$

where s is the number of coupling constraints. Let $\mathbf{x}^{*k}$ denote the optimal solution.

Step 2. If $t(\mathbf{x}^{*k})_j \le \varepsilon^{opt}$, j = 1, 2, ..., s, then stop (optimal solution has been obtained). Otherwise, (a) let $\mathbf{x}^{k+1} = \mathbf{x}^{*k}$, (b) update the penalty parameters μ and ε, (c) set k=k + 1, and proceed from Step 1.

The efficient solution of the problem in Step 1 is discussed in section 6.5.

6.4.3.1 Theoretical issues

The convergence of the above algorithm is proved in Zenios, et al., (1990) and can be summarized as follows.

Associate the multipliers λ with constraints $\sum_{i=1}^{n} \mathbf{B}_i \mathbf{x}_i = \mathbf{b}_0$, $\omega \ge \mathbf{0}$ with constraints $\mathbf{Ex} \le \mathbf{b}$, and $\theta^u, \theta^l \ge \mathbf{0}$ with constraints $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$, and let $\mathbf{g} = \nabla \mathbf{f}$.

The first-order optimality conditions for problem (6.4.12) are then:

$$\mathbf{g}(x) + \mathbf{A}^T\lambda + \mathbf{E}^T\omega + \theta^u - \theta^l = \mathbf{0}$$

$$\sum_{i=1}^{n} \mathbf{B}_i \mathbf{x}_i = \mathbf{b}_0$$

$$
\begin{aligned}
&\mathbf{Ex} \le \mathbf{b} \\
&\mathbf{l} \le \mathbf{x} \le \mathbf{u} \\
&\omega \ge \mathbf{0};\ \theta^{\mathbf{u}} \ge \mathbf{0},\ \theta^{\mathbf{l}} \ge \mathbf{0} \\
&\omega(\mathbf{Ex} - \mathbf{b}) = \mathbf{0} \\
&\theta^{\mathbf{u}}(\mathbf{x} - \mathbf{u}) = \mathbf{0} \\
&\theta^{\mathbf{l}}(\mathbf{x} - \mathbf{l}) = \mathbf{0}
\end{aligned}
$$

Vectors $\mathbf{x}^*$, ω^*, λ^*, θ^* satisfying the above, are unique if one assumes that strict complementarity and second-order sufficiency hold. For the optimality conditions of (6.4.14), if one associates λ_ε with constraints $\sum_{i=1}^{n} \mathbf{B}_i \mathbf{x}_i = \mathbf{b}_0$ and $\theta_\varepsilon^{\mathbf{u}}$, $\theta_\varepsilon^{\mathbf{l}} \ge \mathbf{0}$ with constraints $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$, the following is obtained.

$$
\begin{aligned}
&\mathbf{g}(\mathbf{x}) + \mu \sum_{j=1}^{s} \nabla\phi(\varepsilon, \mathbf{t}_j) + \mathbf{A}^{\mathbf{T}}\lambda_\varepsilon + \theta_\varepsilon^{\mathbf{u}} - \theta_\varepsilon^{\mathbf{l}} = \mathbf{0} \\
&\sum_{i=1}^{n} \mathbf{B}_i \mathbf{x}_i = \mathbf{b}_0 \\
&\mathbf{l} \le \mathbf{x} \le \mathbf{u} \\
&\theta_\varepsilon^{\mathbf{u}} \ge \mathbf{0},\ \theta_\varepsilon^{\mathbf{l}} \ge \mathbf{0} \\
&\theta_\varepsilon^{\mathbf{u}}(\mathbf{x} - \mathbf{u}) = \mathbf{0} \\
&\theta_\varepsilon^{\mathbf{l}}(\mathbf{x} - \mathbf{l}) = \mathbf{0}
\end{aligned}
$$

Letting $\mathbf{A}(\mathbf{x}) = \{j | 0 < \mathbf{t}_j \le \varepsilon\}$ be the set of active constraints and $\mathbf{V}(\mathbf{x}) = \{j | \mathbf{t}_j > \varepsilon\}$ be the set of violated constraints, the first of the optimality conditions of problem (6.4.14) reduces to

$$
\mathbf{g}(\mathbf{x}_\varepsilon) + \frac{\mu}{\varepsilon} \sum_{j \in \mathbf{A}(\mathbf{x}\varepsilon)} (\mathbf{E}_j \mathbf{x}_\varepsilon - \mathbf{d}_j)\mathbf{E}_j^{\mathbf{T}} + \mu \sum_{j \in \mathbf{V}(\mathbf{x}\varepsilon)} \mathbf{E}_j^{\mathbf{T}} + \mathbf{A}^{\mathbf{T}}\lambda_\varepsilon + \theta_\varepsilon^{\mathbf{u}} - \theta_\varepsilon^{\mathbf{l}} = \mathbf{0}
$$

where $\mathbf{x}_\varepsilon$ denotes a solution to the optimality conditions of the penaltized problem (6.4.14). Now let

$$
(\omega_\varepsilon)_j = \begin{cases} \mu & \text{for } j \in \mathbf{V}(\mathbf{x}_\varepsilon), \\ \dfrac{\mu(\mathbf{E}_j \mathbf{x}_\varepsilon - \mathbf{b}_j)}{\varepsilon} & \text{for } j \in \mathbf{A}(\mathbf{x}_\varepsilon), \\ 0 & \text{otherwise.} \end{cases}
$$

then the above reduces to

$$g(x_\varepsilon) + E^T\omega_\varepsilon + A^T\lambda_\varepsilon + \theta^u_\varepsilon - \theta^l_\varepsilon = 0,$$

which is reminiscent of the first of the optimality conditions of problem (6.4.12). Thus a solution x_ε to the penalty problem (6.4.14) and the multipliers ω_ε, λ_ε, θ^u_ε, θ^l_ε defined above, satisfy the conditions for optimality of the original problem.

6.4.3.2 Implementation Issues

In Step 1 the coupling constraints are eliminated by the use of a penalty function. The penalty is controlled by the parameters μ and ε and the violation of the coupling constraints. In the limit ($Lim_{\eta\to\infty}\ \mu = \infty$), the violation of the coupling constraints tends to 0, forcing the penalty term $\mu \sum_{j=1}^{s} \phi(\varepsilon, t_j)$ in the objective function to 0 as well. At this point the solution of the quadratic penalty program is the optimal solution of the original linear problem.

The updating of the penalty parameter μ and the accuracy parameter ε in Step 1 is performed with respect to the sign of the difference between the violations of the coupling constraints calculated for the solution obtained in the current iteration and ε by the rule:

If all current violations are less than ε

$$\varepsilon^{k+1} = \max\{\varepsilon^{opt}, \eta_1\varepsilon^k\}$$

else

$$\mu^{k+1} = \frac{\mu^k}{\eta_2\varepsilon^k} \max_{j \in V(x^k)} t_j,$$

where $\eta_1, \eta_2 \in (0, 1]$, and V is the set of all coupling constraints for which the violation calculated on respect of the current vector x^k is greater than ε.

The above rule can be justified as follows.

Case 1: If $V(\mathbf{x}^k) = \varnothing$, this is an indication that the magnitude of the penalty parameter μ was adequate in the previous iteration since ε-feasibility is achieved ($\sum_{i=1}^{n} \mathbf{B}_i \mathbf{x}_i - \mathbf{b}_0 \leq \varepsilon$). In this case the infeasibility tolerance parameter ε may be reduced.

Case 2: If $V(\mathbf{x}^k) \neq \varnothing$, the current point is not ε-feasible, an indication that the penalty parameter m should be increased. The increase is chosen proportionately to the degree of infeasibility.

The algorithm, as described above, solves decomposable linear problems with coupling constraints in inequality form. To use the algorithm for problems of the form (6.2.2) the linear-quadratic penalty function ϕ was modified to the one given below.

$$\phi_1(\varepsilon,\mathbf{t}) = \begin{cases} 0 & \text{if } |\mathbf{t}| \leq \varepsilon^{opt}, \\ \dfrac{\mathbf{t}^2}{2\varepsilon} & \text{if } \varepsilon^{opt} < |\mathbf{t}| \leq \varepsilon, \\ \mathbf{t} - \dfrac{\varepsilon}{2} & \text{if } |\mathbf{t}| \geq \varepsilon, \end{cases}$$

The penalty function ϕ_1 is based on the principals of ϕ but, since the coupling constraints are in equality form, the focus is now in the sign of the difference between the absolute value of the violation of the coupling constraints and the infeasibility tolerance ε.

In Step 0 of the algorithm the decomposable structure of the constraint matrix can be exploited. The objective function is linear and therefore separable and the problem can be decomposed into independent subproblems, which could be solved in parallel. Any of the algorithms described in Chapter 4 could be used for the solution of the independent subproblems. The same is also valid for the problems generated in Step 1 of the algorithm if they are linear, i.e., if the violation t_j of the coupling constraints is less than ε^{opt} or greater than ε. If the problems generated are quadratic ($\varepsilon^{opt} \leq |t| \leq \varepsilon$), then the objective function is, in general, non-separable and non-convex. If the problem was convex one could still exploit the special feature of the constraint matrix using the transformation utilized in the unified dual ascent algorithm for non-separable quadratic programming. The matrix containing the

coefficients of the quadratic and mixed terms of a non-convex quadratic problem is non-positive definite and there is no valid value for the positive parameter δ, as described in § 5.8.3. In the following section of this chapter an approach, which takes into account the decomposable structure of the constraint matrix, for the solution of such problems will be described.

6.5 Linearization via Simplicial Decomposition

In this section the implementation of an approach, based on simplicial decomposition, for the solution of the quadratic non-separable subproblems generated by the decomposition algorithm of § 6.4.3 is described.

The simplicial decomposition algorithm was first proposed in Holloway, (1974) as an extension to the linearization technique of Frank-Wolfe. Significant enhancements were added by Von Hohenbalken, (1977). In Hearn, et al., (1987) a memory-efficient variant of the algorithm was developed. Mulvey, et. al., (1990) developed an inexact variant specialized for network structures.

Simplicial decomposition iterates by solving a sequence of linear problems to generate vertices of a polytope $\mathbf{X}$, [Pinar & Zenios, (1992)]. A non-linear (quadratic) master problem optimizes the penaltized function $\Phi_{(\mu,\varepsilon)}$ on the simplex specified by the vertices generated by the subproblems. The simplicial decomposition at the k-th iteration of the linear-quadratic penalty decomposition algorithm is stated as follows.

Step 0. (a) Set $\nu = 0$, and (b) use $\mathbf{z^0} = \mathbf{x^k} \in \mathbf{X} \equiv \{\mathbf{x} \mid \mathbf{diag}[\mathbf{A_1}\ \mathbf{A_2}, ..., \mathbf{A_n}]\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$ as the starting point. (c) Let $\mathbf{Y} = \mathbf{\emptyset}$, and (d) $\upsilon = 0$ denotes the set of generated vertices and their number, respectively.

Step 1. (Linearized subproblem). (a) Compute the gradient of the penalty function $\Phi_{(\mu,\varepsilon)}$ at the current iterate z^ν and (b) solve a linear program to get a vertex of the constraint set, i.e., solve for $\mathbf{y}^* = \text{argmin}_{\mathbf{y}\in\mathbf{X}}\ \mathbf{y}^T\nabla\Phi_{(\mu,\varepsilon)}(\mathbf{z}^\nu)$ and (c) let $\mathbf{Y} = \mathbf{Y} \cup \{\mathbf{y}^*\}$, (d) $\upsilon=\upsilon+1$.

Step 2. (Non-linear master problem). Using the set of vertices $\mathbf{Y}$ to represent a simplex contained in the constraint set $\mathbf{X}$, (a) find the optimizer of the penaltized objective function $\Phi_{(\mu,\varepsilon)}$ over this subset of $\mathbf{X}$. (b) Let $\mathbf{w}^* = \text{argmin}_{\mathbf{w}\in \mathbf{W}_\upsilon} \Phi_{(\mu,\varepsilon)}(\mathbf{Bw})$ where $\mathbf{W}_\upsilon = \{\mathbf{w_i} \mid \sum_{i=1}^{\upsilon} w_i = 1,\ w_i \geq 0\ \forall\ i = 1, 2, ..., \upsilon\}$ and $\mathbf{B} = [\mathbf{y^1}|\mathbf{y^2}|...|\mathbf{y^\upsilon}]$ is the basis for the simplex generated by the set of vertices $\mathbf{Y}$. The optimizer of $\Phi_{(\mu,\varepsilon)}$ over the simplex is given by $\mathbf{z^{v+1}} = \mathbf{Bw}^*$.

Step 3. Let v = v + 1, and return to Step 1.

The special features of Steps 1 and 2 of the above algorithm will be commented in the following section.

In Step 1 the algorithm solves a linear approximation to the quadratic problem. Because of the decomposable structure of the set $\mathbf{X}$ the problem can be solved independently for each block of the matrix $\mathbf{diag}[\mathbf{A_1}\ \mathbf{A_2}\ ...\ \mathbf{A_n}]$

Step 1 (Decomposed linear subproblems)

For each i = 1, 2, ..., n solve

$$\begin{aligned} \mathbf{Min}\ & \mathbf{y_i}^T \nabla_i \Phi_{(\mu,\varepsilon)}(\mathbf{z^v}) \\ \mathbf{s.t.}\ & \mathbf{A_i x_i} = \mathbf{b_i} \\ & \mathbf{y_i} \geq \mathbf{0} \end{aligned}$$

The non-linear (quadratic) master problem of Step 2 of the simplicial algorithm is of much smaller size than the original problem generated by the decomposition algorithm since it is posed as a problem over the weights $\mathbf{w}$ associated with the vertices. Furthermore, it has a very simple constraint structure: non-negativity constraints and a simplex equality constraint. The single constraint master program can be written in the form:

$$\begin{aligned} \mathbf{Min_w}\ & \Phi_{(\mu,\varepsilon)}(\mathbf{Bw}) \\ \mathbf{s.t.}\ & \sum_{i=1}^{\upsilon} w_i = 1 \\ & \mathbf{w} \geq \mathbf{0}, \qquad i = 1, 2, ..., \upsilon \end{aligned}$$

Pinar and Zenios, 1992 used the same simplicial decomposition approach to overcome the difficulty of solving non-linear problems of non-separable objectives but they

solved the master problem with pure non-linear techniques. That approach requires the computation of a descent direction and the performance of a line-search in each iteration.

In the case of linear programming problems, which is of interest in this chapter, the objective function of the master problem is quadratic. As an alternative to the non-linear techniques, it was proposed that the same algorithm for single constraint quadratic problems as in the implementation of the unified dual ascent algorithm in Chapter 5 be used.

The advantage of this approach is that an iteration of a non-linear technique is, in general, more expensive than that of the specialized algorithm for quadratic problems with one constraint because no line-search is performed.

6.6 An Improved Linear-Quadratic Penalty Function

The smoothed linear-quadratic penalty function used in § 6.4.3 to eliminate the coupling constraints can be improved by modifying both the linear and the quadratic terms of ϕ_1.

The linear-quadratic penalty function proposed here is of the form:

$$\phi_2(\varepsilon,t) = \begin{cases} 0 & \text{if } |t| \le \varepsilon^{opt}, \\ \left(\dfrac{t}{\sqrt{\varepsilon}} - \dfrac{\sqrt{\varepsilon}}{4}\right)^2 & \text{if } \varepsilon^{opt} < |t| \le \varepsilon, \\ \dfrac{t}{2} + \dfrac{\sqrt{\varepsilon}}{8} & \text{if } |t| \ge \varepsilon, \end{cases}$$

The above function differs from ϕ_1 because the minimum of the quadratic term of ϕ_2 is not achieved in the point $t = 0$ but in $t = \frac{\varepsilon}{4}$. For values of $t < \frac{\varepsilon}{4}$ penalties greater than these of function ϕ_1 are added to the objective function. In fact, penalties increases as $t \rightarrow 0$ and $t < \frac{\varepsilon}{4}$. If optimality is not achieved before t becomes less than $\frac{\varepsilon}{4}$ then, in general, the decomposition algorithm will fail to converge. To prevent that happening one must ensure that the starting value of ε is small enough so that optimality is achieved before t becomes

less that $\frac{\varepsilon}{4}$. One way to do that is to start the decomposition algorithm with a good estimate of the solution vector **x.** If **x** is close to the optimal solution of the problem then the maximum value of vector $\mathbf{t} = \mathbf{Bx} - \mathbf{b_0}$, which is equal to ε_0, will be relatively small and hopefully close enough to ε^{opt}. In order to satisfy that restriction one must solve the relaxed from coupling constraints problem and use the result as a starting vector instead of an arbitrary one. It could be argued that solving this problem results in increasing the CPU time required by the algorithm but experimental results do not support that argument. This is due to the fact that the problem solved is separable and thus the subproblems could be solved in parallel using any of the algorithms described in Chapter 4. Also, function ϕ_2 introduced in this study converges faster than ϕ_1 which results in a big reduction of the iterations required by the algorithm to converge.

6.7 Computational Experience

In this section numerical results on the performance of the described decomposition algorithms on a set of random LP problems will be reported. For comparison purposes the same problems were solved by an interior point method presented in Chapter 4. The method chosen was the predictor corrector method and the CPU time and number of floating point operations required by the algorithm can be found under the column interior point method of Table 6.1. The tests were carried out in MATLAB, ver. 4.2 on a networked SPARC workstation. The number of floating point operations recorded was obtained by the FLOPS function of MATLAB. The functions CLOCK and ETIME were used to obtain the sequential CPU time required for each implementation.

The values reported under parallel time are obtained by simulating parallelism. It is assumed that the number of processors available is equal to the number of subproblems in each structured problem. When the independent subproblems, linear or quadratic, are solved, the time required for the solution of each one of them is recorded and the maximum

of these times is used as the parallel time required for the solution of these subproblems in each iteration. The parallel time reported is the sum of the parallel times computed for each iteration of the algorithm plus the time required for the algorithm to perform sequential operations. Parallelism is simulated in the simplicial decomposition algorithm and in the initial phase of the linear-quadratic penalty decomposition approach using the penalty function ϕ_2 where linear problems can be solved in parallel. In the decentralized algorithm and the decomposition approach based on Augmented Lagrangian parallelism can be simulated in the solution of the quadratic subproblems generated in each iteration of these algorithms.

The test problems were based on those used by Mangasarian, (1985) and Salhi, (1989). These problems were generated as follows. The constraint matrix **A** was fully dense with random elements a_{ij} uniformly distributed in the interval [–100, 100]. The right hand side was chosen so that

$$b_i = \begin{cases} \sum_{j=1}^{n} a_{ij} & \text{if } a_{ij} > 0 \\ -1 + 2\sum_{j=1}^{n} a_{ij} & \text{if } \sum_{j=1}^{n} a_{ij} \le 0 \end{cases} \quad , i = 1, 2, \ldots, m$$

and the cost vector

$$c_j = \sum_{i \in J} a_{ij}, \text{ where } J = \left\{ i \mid \sum_{j=1}^{n} a_{ij} > 0 \right\}, j = 1, 2, \ldots, n.$$

In these tests the above problems constitutes blocks linked with a set of rows also randomly generated to form the structured problems. However, vectors **b** and **c** of the structured problems comply with their above definition, taking the linking rows into account. Point **e** is, therefore, primal optimal. A sample of a randomly generated, 2-block, LP problem is given in Figure 6.3.

The convergence for this set of problems was monitored by the difference between the current prediction of the solution vector **x** and a suitable (in size) unit vector. The convergence criterion was set as follows.

$\max(\text{abs}(\mathbf{x} - \text{ones}(\text{size}(\mathbf{x})))) < \varepsilon^{opt}$

c:	65.7118	184.2489	259.3944	316.1557	187.3205	b:
A:	21.8959	67.8865	93.4693			183.2517
	4.7045	67.9296	38.3502			110.9843
				51.9416	5.3462	57.2878
				83.0965	52.970	136.0666
				3.4572	67.1149	70.5721
Linking	0.7698	6.6842	68.6773	93.0436	52.6929	221.8678
Rows	38.3416	41.7486	58.8977	84.6167	9.1965	232.8010

Figure 6. 3 A Sample Randomly Generated Problem with Entries Uniformly Distributed in [0, 100]

In the test performed ε^{opt} was set to 0.01. Problems with 2, 4 and, 8 independent blocks were solved. The constraint matrix of each subproblem was a fully dense matrix of size 5×10. To study the effect the number of linking constraints has on the solution process of the described decomposition approaches two variants of each problem were solved. In the first experiment only two linking constraints were added to the decomposable block of constraints. In the second test the number of linking constraints was set to 20. The results obtained from these test problems are recorded in Tables 6.1 and 6.2. The time ratio reported in Table 6.2 shows the speed up of the solution process in a simulated parallel environment.

The simplicial decomposition algorithm used in the implementations of the linear-quadratic penalty decomposition approach for both the penalty functions ϕ_1 and ϕ_2 requires the solution of linear problems. Linear problems are also solved in the initialization phase of the algorithm when the penalty function is ϕ_2 used. For the solution of these problems the predictor corrector algorithm of Chapter 4 was used.

For the solution of the quadratic subproblems generated in each iteration of the decentralized algorithm and the Augmented Lagrangian method the unified dual ascent algorithm described in Chapter 5 was used.

LP	Linear-Quadratic Penalty Funct - ϕ_2			Linear-Quadratic Penalty Funct - ϕ_1			Decentralized Algorithm			Augmented Lagrangian			Interior Point Method	
B_L	Ite	Time	Flops	Ite	Time	Flops	Ite	Time	Flops	Ite	Time	Flops	Time	Flops
2_2	1	1.432	28741	4	57.68	2627440	3	2.400	7820	5	14.18	43687	0.059	1875
2_20	1	1.650	38561	5	90.03	3341448	4	4.246	20396	5	14.42	80515	0.262	5843
4_2	1	2.459	41227	5	134.4	11715957	5	13.97	34650	5	53.07	255453	0.425	10076
4_20	1	8.289	49585	6	174.0	14518127	6	12.17	79329	7	82.98	555127	0.479	22929
8_2	1	29.80	124642	5	367.3	52344499	6	20.32	151756	5	497.7	2606878	0.882	58990
8_20	1	31.27	130077	6	491.1	69756213	6	22.86	172555	7	586.1	5635317	0.987	61101

Table 6.1 Comparative Results of Decomposition Approaches

LP	Linear-Quadratic Penalty Function - ϕ_2			Linear-Quadratic Penalty Function - ϕ_1			Decentralized Algorithm			Augmented Lagrangian		
B_L	Time Sequen	Time Parallel	Time Ratio	Time Sequen	Time Parallel	Time Ratio	Time Sequen	Time Parallel	Time Ratio	Time Sequen	Time Parallel	Time Ratio
2_2	1.432	1.041	1.377	57.68	52.70	1.094	2.400	1.511	1.562	14.18	7.31	1.960
2_20	1.650	1.216	1.356	90.03	83.49	1.078	4.246	2.443	1.724	14.42	7.462	2.000
4_2	2.459	1.485	1.655	134.4	123.3	1.089	13.97	6.38	2.222	53.07	14.92	3.556
4_20	8.289	4.689	1.769	174.0	159.0	1.095	12.17	5.27	2.325	82.98	28.53	2.941
8_2	29.77	19.81	1.503	367.3	344.0	1.067	20.32	4.18	4.856	497.7	96.12	5.178
8_20	31.28	23.38	1.351	491.1	460.4	1.066	22.86	4.61	5.000	586.1	89.73	6.531

Table 6.2 Comparative Sequential and Simulated Parallel CPU Time Results

From the results of Table 6.1 the following observations are apparent:

(i) The number of iterations required by all the methods is, in general, small and it was never greater than 7 in the test problems used. For the problems with only two coupling constraints the decomposition approach based on linear-quadratic penalty function ϕ_2 requires only one iteration.

(ii) The decentralized algorithm and the decomposition method based on the linear-quadratic penalty function ϕ_2 are the approaches that require the smallest number of floating point operations. Note that the decentralized algorithm is started with an arbitrary vector. On the other hand the use of the function ϕ_2 requires additional work for the generation of a good estimate for the solution vector.

(iii) Increasing the number of coupling constraints affects, in general, an increase in both iterations and number of floating point operations.

(iv) All decomposition approaches are outperformed by the predictor corrector algorithm which was chosen as one of the best performing representatives of interior point methods.

(v) The time results recorded do not always coincide with the number of floating point operations required by each algorithm. This is mainly due to the fact that all the tests were carried out on a networked computer for which the processing speed is affected by the current workload on the network.

(vi) The cheapest, in floating point operations, iterations are these of the decentralized algorithm followed by those of the method based on augmented Lagrangian. The decomposition approach using linear-quadratic penalty functions have the highest cost per iteration.

Studying the results of Table 6.2 one can conclude that:

(i) The highest speed ups from the simulation of parallel processing in the solution process are obtained by the decomposition approach based on augmented Lagrangian followed by the decentralized algorithm.

(ii) As expected, the savings in CPU time required for the above two algorithms is increased with the number of independent blocks in the constraint matrix of the structured problem.

(iii) Only modest improvements were observed in the simulated parallel environment for the decomposition approach based on linear-quadratic penalty functions. This is

due to the fact that parallelism is only involved in the solution of linear problems which are, in general, easy to solve and cheap both in CPU time and floating point operations.

6.8 Conclusions

It appears from the result of Table 6.1 than none of the decomposition algorithms tested is comparable, in either CPU time or floating point operations, to the selected interior point method for the tests performed. The explanation lies in the nature of the test problems. Larger and more ill-conditioned problems need to be tested because it is in this class of problems where the decomposition techniques are expected to show their advantages and outperform interior point methods.

As expected parallelism did not improve the performance of the linear-quadratic penalty algorithm. In a parallel environment considerable savings in the CPU time required by this method can be expected only for structured problems with a large number of big independent blocks.

The poor performance of the method of Mulvey and Ruszczynski is mainly due to the slow convergence observed in the inner iteration of the algorithm. The number of iterations required for the achievement of optimality in the inner iteration of the algorithm grows as the number of independent blocks increases. Computational experience with that method shows that convergence is achieved even if the inner circle is not complete. For the problems tested the number of iterations of the inner loop was limited to 20. The truncated version performed well and accurately for the test problems of this chapter. A way to improve the performance of the algorithm can also be found in Step 1b of the inner loop. The update of the solution vector performed in this step is based on taking a step along the direction formed on the difference of the current and the previous iterate. An alternative approach, which could be proved efficient, is to perform the updating procedure based on the direction of the gradient of the objective function at the current point.

One of the best performing algorithms is based on the decentralized method. The use of the partial inverse method, [Spingarn, (1985)], yields separable augmented Lagrangians and there is no need for the solution of any kind of master problems.

The most demanding, both in CPU time and floating point operations, method is the one based on the linear-quadratic penalty function ϕ_1. Most of the processing time required by the algorithm is consumed by the simplicial-decomposition algorithm for the solution of the non-separable quadratic master problem even though the number of iterations of the simplicial-decomposition algorithm was limited to 20 in the tests performed.

The inexact solution of the quadratic subproblems generated in each iteration of the implemented algorithms is an attractive feature not only for the methods considered in this chapter but for all decomposition techniques in general. The principle of inexact solution of the master problem first appears in Dantzig and Wolfe, (1960). The idea is that, since these methods are self correcting, the solution obtained after a limited number of iterations can be used as a search direction only. In the experiments carried out for this chapter, variants of the algorithms which limit the number of iterations to 10 and 5 were also tested. Experimental results showed that for less than 20 iterations the obtained solution was not good enough as a prediction of the search direction which results in increasing number of outer loops required for the convergence of the algorithm.

Further research is required for the two most promising algorithms tested, namely the decentralized method and the approach based on linear-quadratic functions. The use of different projections could be considered in the implementation of the decentralized algorithm, [Spingarn, (1985)]. Additional studies should also investigate all the features of the linear-quadratic penalty function ϕ_2 and explain the performance of the algorithm when this function is used. Finally, alternative ways to the simplicial decomposition approach for the solution of the quadratic penalty master problem could be worth investigating. For example, separability can also be exploited when successive linear programming is used or by combining a truncated Newton method with block-partitioning techniques, [Zenios & Pinar, (1992)], as mentioned in Zenios, et al., (1990).

Chapter 7

Conclusions and Further Development

The primary focus of this research was to determine the most effective algorithm for the solution of sparse linear programming problems. These problems may often have some special structure.

The infeasible method has the best performance among the interior point algorithms tested in this research.

The decentralized algorithm is the best performing decomposition algorithm for structured problems and is based on the partial inverse method.

Interior point methods can also be applied to quadratic problems, which are usually generated as subproblems to decomposition techniques for the solution of structured linear programming problems. From the tested algorithms for quadratic programming the approach based on conjugate gradients generally performs better for almost all problems.

At the beginning of this thesis the initial steps of linear programming, along with the latest approaches in the field of interior point methods, were discussed. The algorithm of Karmarkar was described and the principles of the penalty and barrier functions approach were reviewed.

For the efficient implementation of the Karmarkar algorithm or any other interior point method, advanced least square techniques are required. In this respect, this topic was also reviewed and several methods for the solution of the least square problem were presented. The conjugate gradient method was chosen as the most powerful tool among these different approaches.

Preconditioning can be seen as a way to improve the condition number of a matrix and accelerate the convergence of the conjugate gradient method. In order to improve the

performance of the implementation of the conjugate gradient method different preconditioners were considered. Problems with different levels of density and condition numbers were tested and numerical results were reported.

A variant of the projective algorithm was tested against three more interior point methods. The tested algorithms are all based on different approaches. The comparative numerical study of these algorithms, for which sparsity was taken into account, resulted in some interesting conclusions. Test problems were chosen from three different and hard to solve classes of linear programming problems. These were the Hilbert-type, the Klee-Minty and the linear ordering problems. Tests were also carried out on randomly generated problems. Throughout the experiments performed it was confirmed that interior point algorithms preserve their attractive feature of low iteration count and that the level of density has a strong effect on the performance of the different approaches considered. The performance of the tested algorithms as discussed in Chapter 4 allows for the following conclusions to be made. The results for the non-random generated problems of Linear Ordering and Klee-Minty class support the conclusion that the infeasible method performs better than the other algorithms on sparse problems. For classes of denser problems, Hilbert type and randomly generated problems, the Barnes algorithm performs better.

The study of decomposition as a strategy for reducing the work required for the solution of large-scale linear programming problems, constitutes one of the main objectives of this thesis and introduces improvements to the current methods. Structured linear programming problems form an important class of problems frequently occurring in real applications. Large-scale programs almost always have some distinctive structure, thus the use of interior point methods combined with decomposition techniques is not only well justified but also worth investigating. Three different but promising of the most recent approaches on decomposition were studied, implemented and tested, on randomly generated problems. Improvements to some of the algorithms were suggested with significant success. A practical implementation of the simplicial decomposition algorithm was developed and used as a way to exploit separability in one of the decomposition techniques considered. The experience gathered from the numerical study of these algorithms revealed some interesting

features of the tested methods and their components. The best performing algorithm is the one based on the partial inverse method because it leads to totally decentralized problems. The absence of any kind of master problem in the solution process of this approach gives a great advantage to that algorithm because, as mentioned in Chapter 6, even if the master problem is small, its solutions can be expensive due to ill-conditioning. As an example, the simplicial decomposition subroutine consumes most of the CPU time required for the solution of the problem in the linear-quadratic penalty approach.

For the efficient implementation of the decomposition methods mentioned above, advanced convex quadratic programming algorithms are required. In view of this, a survey of recent literature published in this area of algorithms was conducted and three methods that looked promising were selected. The implementation of a specialized algorithm for single constraint quadratic programs was necessary for the development of an efficient variant of one of the methods. The implemented algorithms were tested on a wide range of problems with matrices of different condition numbers. The effect of sparsity was also studied as sparsity was taken into account in the implementation of the algorithms. New parameters that dramatically improve the performance of one of the algorithms were suggested. The comparative results of these algorithms, reported in Chapter 5, indicate that the approach based on conjugate gradients generally performs better for almost all problems. For problems where the matrix $\mathbf{Q}$ is very sparse (12%), though, this algorithm is outperformed by the interior method of Golfarb and Lui. Finally, the unified dual ascent algorithm proved to be a very promising approach but only for problems with separable objectives.

Among the aspects of decomposition techniques discussed in this work, the effect the number of linking constraints has on the solution process is, without doubt, the topic that needs to be further studied. In the tests performed it was assumed that the blocks were weakly linked by only 2 or 20 linking constraints. It still remains to be seen how the link affects the overall work involved in each approach and in what proportion.

Further research is also required for two of the algorithms tested in Chapter 6, namely the decentralized method and the approach based on linear-quadratic functions. The use of different projections and penalty functions could be considered in the implementation

of the decentralized algorithm and the linear-quadratic approach, respectively. Additional studies should also investigate all the features of the linear-quadratic penalty function ϕ_2 and explain the performance of the algorithm when this function is used. Finally, alternatives to the simplicial decomposition approach for the solution of the quadratic penalty master problem could prove to be worth investigating.

Decomposition is also attractive from the point of view of parallelism or concurrency. In Chapter 6 it is shown that favourable structure present in large LP problems may be used to advantage and that all decomposition approaches lend themselves readily to parallel processing. Although parallel time was simulated in the tests performed for this study it would be interesting to see how these algorithms perform in a real parallel or concurrent environment.

Tests on large-scale and ill-conditioned problems with a large number of independent blocks should also be considered. It is in these problems where decomposition techniques are expected to show their advantages against interior point methods.

The unified dual ascent algorithm discussed in Chapter 5 is also a candidate for future investigation. The method seems very promising for separable quadratic problems but its performance is very poor for problems with non-separable objective functions. Indeed, the future of the algorithm would be rather bleak if it is found that the proposed transformation, [Lin & Pang, (1987)], can't be improved and the method is unsuitable for the solution of large real world problems. Finally, an implementation of a quadratic programming algorithm based on the predictor corrector approach as investigated and analysed in § 5.10, would give a clear idea of how this robust and promising algorithm perform on problems with quadratic objective functions.

From these investigations, it appears that decomposition techniques are a serious alternative to interior point methods for large-scale programming. However, the questions raised here need to be answered and more research needs to be carried out before all features of these methods are fully understood.

References

Adler, I., N. K. Karmarkar, M. G. C. Resende and G. Veiga (1989), "An Implementation of Karmarkar Algorithm for Linear Programming." Mathematical Programming, **44**, pp. 297–335.

Andersen, K. D. (1993), "An Infeasible Dual Affine Scaling Method for Linear Programming.", Proceedings of the Scandinavian Workshop in Linear Programming, Denmark.

Anstreicher, K. M. (1989), "A Combined Phase I-Phase II Projective Algorithm for Linear Programming." Mathematical Programming, **43**, pp. 209–223.

Avis, D. and V. Chvatal (1978), "Notes on Bland's Pivoting Rule." Mathematical Programming, **8**, pp. 24–34.

Avriel, M. (1976), Non-linear Programming: Analysis and Methods, Prentice-Hall, Englewood Cliffs, NJ.

Axelsson, O. and G. Lindskog (1986), "On the Rate of Convergence of the Preconditioned Conjugate Gradient Method." Numerische Mathematik, **48**, pp. 499–523.

Barnes, E. R. (1986), "A Variation on the Karmarkar's Algorithm for Linear Programming Problems." Mathematical Programming, **36**(2), pp. 174–182.

Barnes, E. R., S. Chopra and D. L. Jensen (1988), "A Polynomial Time Version of the Affine Scaling Algorithm", Technical Report, 88-101, Graduate School of Business Administration, New York University.

Bayer, D. A. and J. C. Lagarias (1989), "The Non-Linear Geometry of Linear Programming I, Affine and Projective Scaling Trajectories." Transactions of the American Mathematical Society, **314**, pp. 499–526.

Bertsekas, D. P. (1975), "Nondifferentiable Optimization via Approximation." Mathematical Programming Study, **3**, pp. 1–25.

Bertsekas, D. P. (1982), Constrained Optimization and Lagrange Multiplier Methods, Academic Press, New York.

Bland, R. G. (1977), "New Finite Pivoting Rules for the Simplex Method." Mathematics of Operations Research, **2**, pp. 103–107.

Brucker, P. (1984), "An 0(n) Algorithm for Quadratic Knapsack Problems." Operations Research Letters, **3**(3), pp. 163–166.

Burrett, C. M. (1994), "Quadratic Programming", MSc Thesis, Aston University.

Carpenter, T. J., I. L. Lustig, J. M. Mulvey and D. F. Shanno (1993), "Higher Order Predictor-Corrector Interior Point Methods with Application to Quadratic Objectives." SIAM Journal of Optimization, **3**, pp. 696–725.

Carpenter, T. J. and D. F. Shanno (1993), "An Interior Point Method for Quadratic Programs Based on Conjugate Projected Gradients." Computational Optimization and Applications, **2**, pp. 5–28.

Charnes, A., T. Song and M. Wolfe (1984), "An Explicit Solution Sequence and Convergence of Karmarkar's Algorithm", Research Report CCS 501, 78712-1177(512), Center of Cybernetic Studies, College of Business Administration s.202, The University of Texas at Austin, Texas.

Carrol, C. W. (1961), "The Created Response Surface Technique for Optimising Non-linear Restrained Systems." Operations Research, **9**, pp. 169–184.

Charnes, A., T. Song, and M. Wolfe, (1984), "An Explicit Solution Sequence and Convergence of Karmarkar's Algorithm", Research Report CCS 501, 78712-1177(512), Center of Cybernetic Studies, College of Business Administration s.202, The University of Texas at Austin, Texas.

Choi, I. C., C. L. Monma and D. F. Shanno, (1990), "Further Developments of a Primal Dual Interior Point Method." ORSA Journal of Computing, **2**, pp. 304–311.

Chvàtal, V. (1983), Linear Programming, W.H.Freeman & Co, New York.

Cohen, G. (1978), "Optimization by Decomposition and Coordination: A Unified Approach." IEEE Transactions on Automatic Control, **AC-23**, pp. 222–232.

Cohen, G. (1980), "Auxiliary Problem Principle and Decomposition of Optimization Problems." Journal of Optimization Theory and Applications, **32**, pp. 277–305.

Coleman, T. F. (1986), "A Chordal Preconditioner for Large-scale Optimization", Technical Report, 14853, Computer Science Department, Cornell University, Ithaca, New York.

Concus, P., G. H. Golub and G. Meurant (1985), "Block Preconditioning for the Conjugate Gradient Method." SIAM J. Sci. Stat. Comput., **6**, pp. 220–252.

Cook, S. (1983), "An Overview of Computational Complexity." Communications of the ACM, **26**(6), pp. 401–408.

Courant, R. (1943), "Variational Methods for the Solution of Problems of Equilibrium and Vibrations." Bulletin of the American Mathematical Society, **49**, pp. 1–23.

Dantzig, G. B. (1963), Linear Programming and Extensions, Princeton University Press, Princeton, NJ.

Dantzig, G. B. and P. Wolfe (1960), "Decomposition Principle for Linear Programs." Operations Research, **8**, pp. 101–111.

De Buchet, J. (1971), "How to Take Into Account the Low Density of Matrices to Design a Mathematical Programming Package", Large Sparse Systems of Linear Equations, J.K.Reid (Ed.), Academic Press, New York, pp. 101–118.

Dennis, J. E., A. M. Morchedi and K. Turner (1986), "A Variable-Metric Variant of the Karmarkar Algorithm for Linear Programming", Technical Report, 86-13, Department of Mathematical Science, Rice University, Houston, Texas 77251.

Dodani, M. H. and A. J. G. Babu (1990), "Karmarkar's Projective Method for Linear Programming: a Computational Survey ." Int. J. Math. Educ. Sci. Technol., **21**(2), pp. 191–212.

Duff, I. S., A.M. Erisman, J. K. Reid, (1986), Sparse Matrix Computations, Academic Press, London.

Edmonds, J. (1965), "Paths, Trees and Flowers." Canadian Journal of Mathematics, **17**, pp. 449–467.

Fiacco, A. N. and G. D. McCormick (1986), Non-Linear Programming: Sequential Unconstrained Minimization Techniques, Wiley & Sons Ltd.

Fieldhouse, M. and F. M. Tromans (1985), "Convergence, Scaling and Duality in Karmarkar's Projective Algorithm.", Proceedings of the Symposium on Karmarkar's and Related Algorithms for Linear Programming, Burlington House, Piccadilly, London.

Fletcher, R. and C. M. Reeves (1964), "Function Minimization by Conjugate Gradients." Computer Journal, **7**, pp. 149–154.

Fletcher, R. (1976), "Conjugate Gradient Methods for Indefinite Systems." Lecture Notes Math., **506**, pp. 73–89.

Fletcher, R. (1986), "Recent Developments in Linear and Quadratic Programming", NA/94, Department of Maths Sciences, University of Dundee, Scotland.

Fortin, M. and R. Glowinski, (1983), "On Decomposition-Coordination Methods Using an Augmented Lagrangian", Augmented Lagrangian Methods: Applications to the Numerical Solution of Boundary-Value Problems, M. Fortin and R. Glowinski (Ed.), Amsterdam.

Freund, R. M. (1991), "Theoretical Efficiency of a Shifted-Barrier-Function Algorithm for Linear Programming." Linear Algebra and its Applications, **152**, pp. 19–41.

Frisch, K. R. (1955), "The Logarithmic Potential Method of Convex Programming", University Institute of Economics, Oslo.

Garey, M. R. and D. S. Johnson (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H.Freeman & Company, San Francisco, CA.

Gay, D. M. (1987), "A Variant of Karmarkar's Linear Programming Algorithm for Problems in Standard Form." Mathematical Programming, **37**(1), pp. 81–90.

Geoffrion, A. (1971), "Large-scale Linear and Non-linear Programming", Optimization Methods for Large-Scale Systems with Applications, D. A. Wismer (Ed.), McGraw-Hill, New York.

George, A., and J. W. Liu, (1981), Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Inc., Englewood Cliffs.

Ghellinck, G. and J. Ph. Vial, (1986), "A Polynomial Newton Method for Linear Programming." Algorithmica, **1**, pp. 425–453.

Gilbert, J. R., C. Moler and R. Schreiber, (1992), "Sparse Matrices in MATLAB: Design and Implementation." SIAM Journal of Matrix Analysis and Applications, **13**(1), pp. 333–356.

Gill, P. E., W. Murray, M. A. Saunders, J. A. Tomlin and M. N. Wright (1985), "On Projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projected Method." Mathematical Programming, **36**(2), pp. 183–209.

Gill, P. E., W. Murray, M. A. Saunders and M. N. Wright (1981), Practical Optimization, Academic Press.

Gill, P. E., W. Murray, M. A. Saunders and M. N. Wright (1984), "Sparse Matrix Methods in Optimization." SIAM J. Sci. Stat. Comp., **5**(3), pp. 562–589.

Goffin, J.-L., A. Haurie, J.-P. Vial and D. l. Zhu (1993), "Using Central Prices in Decomposition of Linear Programs." European Journal of Operational Research, **64**, pp. 393–409.

Goffin, J. L. (1988), "Affine and Projective Transformations in Nondifferentiable Optimization." International Series of Numerical Mathematics, **84**, pp. 80–91.

Goffin, J. L., A. Haurie and J. P. Vial (1992), "Decomposition and Nondifferentiable Optimization With the Projective Algorithm." Management Science, **38**(2), pp. 284–302.

Goffin, J. L. and J. P. Vial (1990), "Cutting Planes and Column Generation Techniques With the Projective Algorithm.".

Goffin, J. L. and J. P. Vial (1993), "On the Computation of Weighted Analytic Centers and Dual Ellipsoids With the Projected Algorithm." Mathematical Programming, **60**, pp. 81–92.

Goldfarb, D. and S. Lui (1991), "An $O(n^3L)$ Interior Point QP algorithm." Mathematical Programming, **49**, pp. 325–340.

Goldfarb, D. and S. Mehrotra (1988), "A Relaxed Version of Karmarkar's Method." Mathematical Programming, **40**(3), pp. 289–315.

Goldfarb, D. and S. Mehrotra (1988), "Relaxed Variants of Karmarkar's Algorithm for Linear Programs with Unknown Optimal Objective Value." Mathematical Programming, **40**(2), pp. 183–195.

Golub, G. and C. Van Loan (1983), Matrix Computations, John Hopkins University Press, Baltimore.

Gonzaga, C. C. (1987), "An Algorithm for Solving Linear Programs in $0(n^3L)$ Operations", Technical Report, UCB/ERL 87/10, Electronic Research Lab., University of California, Berkeley.

Grötschel, M., M. Jünger and G. Reinelt (1984), "Optimal Triangulation of Large Real World Input-Output Matrices." Statistische Hefte, **25**, pp. 28–42.

Hearn, D. W., S. Lawphongpanich and J. A. Ventura (1987), "Restricted Simlicial Decomposition: Computation and Extensions." Mathematical Programming Study, **31**, pp. 99–118.

Heath, M. T. (1984), "Numerical Methods for Large Sparse Linear Least Squares Problems." SIAM J. Sci. Stat. Comp., **4**(3), pp. 497–513.

Helgason, R., J. Kennington and H. Lall (1980), "A Polynomially Bounded Algorithm for a Single Constrained Quadratic Program." Mathematical Programming, **18**, pp. 338–343.

Hellier, F. S. and G. J. Lieberman (1986), Introduction to Operations Research, Holden-Day, Inc., Oakland, California.

Hertog, D. and C. Ross (1991), "A Survey of Search Directions in Interior Point Methods for Linear Programming." Mathematical Programming, **52**, pp. 481–509.

Hertog, D. D., C. Roos and T. Terlaky (1992), "On the Classical Logarithmic Barrier Function Method for a Class of Smooth Convex Programming Problems." Journal of Optimization Theory and Applications, **73**(1), pp. 1–25.

Hestenes, M. R. and E. Stiefer (1952), "Methods of Conjugate Gradients for Solving Linear Systems." Journal of Research of the National Bureau of Standards (US), **49**, pp. 409–436.

Holloway, C. A. (1974), "An Extension of the Frank-Wolfe Method of Feasible Directions." Mathematical Programming, **6**, pp. 14–27.

Iri, M. and H. Imai (1986), "A Multiplicative Barrier Function Method for Linear Programming." Algorithmica, **1**, pp. 455–482.

Kantorovich, L. N. (1939), "Mathematical Methods in the Organization and Planning of Production." Translated in Management Science, **6**, pp. 366–422.

Karmarkar, N. (1984), "A New Polynomial-Time Algorithm for Linear Programming." Combinatorica, **4**(4), pp. 373–395.

Karmarkar, N. (1984), "A New Polynomial-Time Algorithm for Linear Programming.", Proceedings of the 16th Annual ACM Symposium on Theory of Computing, Washington D.C.

Karmarkar, N. and K. G. Ramakrishnan (1991), "Computational Results of an Interior Point Algorithm for Large-scale Linear Programming." Mathematical Programming, **52**, pp. 555–586.

Khachyan, L. G. (1979), "A Polynomial-Time Algorithm for Linear Programming." Combinatorica, **4**(4), pp. 191–179.

Klee, V. and G. L. Minty (1972), "How Good Is the Simplex Algorithm?", Equalities III, O. Shisha (Ed.), Academic Press, New York.

Kojima, M., S. Mizuno and A. Yoshise (1989), "A Primal-Dual Interior Point Algorithm for Linear Programming", Progress in Mathematical Programming: Interior Point and Related Methods, N. Megiddo (Ed.), Springer Verlag, New York, pp. 29–47.

Kortanek, K. O. (1993), "Vector-Supercomputer Experiments with the Primal-Affine Programming Scaling Algorithm." SIAM J. Sci. Comput., **14**(2), pp. 279–294.

Kortanek, K. O. and M. Shi (1987), "Convergence Results and Numerical Experiments on a Linear Programming Hybrid Algorithm." European Journal of Operational Research, **32**, pp. 47–61.

Kovacevic-Vujcic, V. V. (1991), "Improving the Rate of Convergence of Interior Point Methods for Linear Programming." Mathematical Programming, **52**, pp. 467–479.

Kronsjö, L. (1985), Computational Complexity of Sequential and Parallel Algorithms, Wiley & Sons Ltd, Chichester.

Kronsjö, L. (1987), Algorithms: Their Complexity and Efficiency, Wiley & Sons Ltd, Chichester.

Lasdon, L. S. (1970), Optimization Theory for Large Systems, Macmillan, Toronto.

Levin, J. A. (1965), "On an Algorithm for the Minimization of Convex Functions." Doklady Akademii Nauk SSSR, **160**(6).

Lin, Y. Y. and J.-S. Pang (1987), "Iterative Methods for Large Convex Quadratic Problems: A Survey." SIAM J. Control and Optimization, **25**(2), pp. 383–411.

Lindfield, G. and J. Penny, (1995), Numerical Methods Using MATLAB, Ellis Horwood, London.

Longley, W. J. (1984), Linear Least Squares Computations Using Orthogonalization Methods, Marcel Dekker, Inc., New York 10016.

Lovász, L. (1980), "The Ellipsoid Algorithm: Better or Worse than the Simplex?" Mathematical Intelligence, **2**, pp. 141–146.

Lovász, L. (1984), "The Mathematical Notion of Complexity.", Proceedings of the 9th Triennial World Congress of IFAC, Budapest.

Lustig, I. J. (1985), "A Practical Approach to Karmarkar's Algorithm", Technical Report, SOL 85-5, Department of Operations Research, Stanford University, Stanford, CA 94305.

Lustig, I. J., R. E. Marsten and D. F. Shanno (1991), "Computational Experience with a Primal-Dual Interior Point Method for Linear Programming." Linear Algebra and its Applications, **152**, pp. 192–222.

Lustig, I. J., R. E. Marsten and D. F. Shanno (1994), "Interior Point Methods for Linear Programming: Computational State of the Art." ORSA Journal on Computing, **6**(1), pp. 1–14.

Mahey, P. and P. D. Tao, (1993), "Proximal Techniques for Large-Scale Linear Programming.", Proceedings of Scandinavian Workshop in Linear Programming, Denmark.

Mangasarian, O. L. (1985), "Iterative Solution of Linear Programs." SIAM Journal of Numerical Analysis, **18**(4), pp. 606–614.

Marsten, R., R. Subramanian, I. Lustig and D. Shanno (1990), "Interior Point Methods for Linear Programming: Just Call Newton, and Fiacco and McCormick!" Interfaces, **20**, pp. 105–116.

McShane, K. A., C. L. Monma and D. F. Shanno (1989), "An Implementation of a Primal-Dual Interior Point Method for Linear Programming." ORSA J. Comput., **1**, pp. 70–83.

Meggido, N. (1986), "Introduction: New Approaches to Linear Programming." Algorithmica, **1**, pp. 387–394.

Megiddo, N. (1989), "Pathways to the Optimal Set in Linear Programming", Progress in Mathematical Programming: Interior Point and Related Methods, N. Megiddo (Ed.), Springer Verlag, New York, pp. 131–138.

Mehrotra, S. (1992), "On the Implementation of a Primal-Dual Interior Point Method." SIAM Journal on Computing, **2**(4), pp. 575–601.

Meyerink, J. A. and H. A. Van der Vorst (1977), "An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-matrix." Math. Comp., **31**, pp. 148–162.

Meyerink, J. A. and H. A. Van der Vorst (1981), "Guidelines for the Usage of Incomplete Decompositions in Solving Sets of Linear Equations as They Occur in Practical Problems." J. Comp. Phys, **44**, pp. 134–155.

Monteino, R. C. and I. Adler (1989), "Interior Path Following Primal-Dual Algorithms. Part I: Linear Programming." Mathematical Programming, **44**, pp. 27–42.

Mulvey, J. M., S. A. Zenios and D. P. Ahlfeld (1990), "Simplicial Decomposition for Convex Generalized Networks." Journal of Information and Optimization Sciences, **11**, pp. 359–387.

Mulvey, J. M. and H. Vladimirou (1991), "Applying the Progressive Hedging Algorithm to Stochastic Generalized Networks." Ann. Operations Research, **341**, pp. 399–424.

Mulvey, J. M. and A. Ruszczynski, (1992), "A Diagonal Quadratic Approximation Method for Large-scale Linear Programs." Operational Research Letters, **21**, pp. 205–215.

Nemhauser, G. L., A. H. G. Rinnooy Kan and M. J. Todd (1989), Handbook in Operations Research & Management Science, Volume 1: Optimization, Elsevier Sciences Publishers B.V., Amsterdam.

Paige, C. C. and M. A. Saunders (1982), "LSQR: An Algorithm for Sparse Linear Equations and Sparse Least-Squares." ACM Transactions on Mathematical Software, **8**, pp. 43–71.

Pang, J.-S. (1983), "Methods for Quadratic Programming: A Survey." Computers and Chemical Engineering, **7**, pp. 583–594.

Papadimitriou, C. H. and K. Steiglitz (1982), Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, NJ.

Pinar, M. C. and S. A. Zenios, (1992), "Parallel Decomposition of Multicommodity Network Flows Using a Linear-Quadratic Penalty Algorithm." ORSA Journal on Computing, **4**(3), pp. 235–249.

Ponnambalam, K., V. H. Quintana and A. Vannelli (1992), "A Fast Algorithm for Power System Optimization Problem Using an Interior Point Method." Transactions of Power Systems, **7**(2), pp. 892–899.

Rockafellar, R. T. (1976), "Monotone Operations and the Proximal Point Algorithm." SIAM Journal of Control and Optimization, **14**(5), pp. 877–898.

Rosen, J. B. (1964), "Primal Partition Programming for Block Diagonal Matrices." Numerische Mathematik, **6**, pp. 250–260.

Ruszczynski, A. (1989), "Au Augmented Lagrangian Decomposition Method for Block Diagonal Linear Programming Problems." Operational Research Letters, **8**, pp. 287–294.

Salhi, A. (1989), "Karmarkar's Algorithm: Extensions and Implementation", Ph.D. Thesis, Aston University.

Saunders, M. A. (1994), "Major Cholesky Would Feel Proud." ORSA Journal on Computing, **6**(1), pp. 23–34.

Setiono, R. (1990), "Interior Dual Proximal Point Algorithm Using Preconditioned Conjugate Gradient", Computer Science Technical, 951, University of Wisconsin, Madison.

Shanno, D. F. (1988), "Computing Karmarkar Projections Quickly." Mathematical Programming, **41**, pp. 61–71.

Shanno, D. F. and R. E. Marsten (1988), "A Reduced-Gradient Variant of Karmarkar's Algorithm and Null-Space Projections." Journal of Optimization Theory and Applications, **57**(3), pp. 383–397.

Simonnard, M. (1966), Linear Programming, Prentice-Hall, Englewood Cliffs, NJ.

Smale, S. (1983), "On the Average Number of Steps of the Simplex Method for Linear Programming." Mathematical Programming, **27**, pp. 241–262.

Spingarn, J. E. (1985), "Applications of the Method of Partial Inverses to Convex Programming: Decomposition." Mathematical Programming, **32**, pp. 199–223.

Taha, H. A. (1992), Operations Research: An Introduction, Macmillan Publishing Company, New York.

Tewarson, R. P. (1973), Sparse Matrices, Academic Press, New York.

Todd, M. J. and B. P. Burrell (1986), "An Extension of Karmarkar Algorithm for Linear Programming Using Dual Variables." Algorithmica, **1**, pp. 409–424.

Todd, M. J. (1993), "Combining Phase-I and Phase-II in a Potential Reduction Algorithm for Linear Programming." Mathematical Programming, **59**, pp. 133–150.

Todd, M. J. and Y. Wang (1993), "On Combined Phase 1-Phase 2 Projective Methods for Linear Programming." Algorithmica, **9**, pp. 64–83.

Todd, M. J. (1994), "Theory and Practice for Interior Point Methods." ORSA Journal on Computing, **6**(1), pp. 28–31.

Tomlin, J. A. (1985), "An Experimental Approach to Karmarkar's Projective Methods for Linear Programming.", Proceedings of the Symposium on Karmarkar's and Related Algorithms for Linear Programming, Geological Society, Burlington House, Piccadilly, London.

Traub, J. F.and Wozniakowki, (1982), "Complexity of Linear Programming." Operations Research Letters, **1**(2), pp. 59–62.

Turner, K. (1987), "A Variable-Metric Variant of the Karmarkar Algorithm for Linear Programming", Technical Report, 87-13, Department of Mathematical Science, Rice University, Houston, Texas 77251.

Van der Sluis, A. (1969), "Condition Numbers and Equilibration of Matrices." Numerische Mathematik, **14**, pp. 14–23.

Van der Sluis, A. and H. A. Van der Vorst (1986), "The Rate of Convergence of Conjugate Gradients." Numerische Mathematik, **48**, pp. 543–560.

Van der Vorst, H. A. and K. Dekker (1988), "Conjugate Gradient Type Methods and Preconditioning." Journal of Computational and Applied Mathematics, **24**, pp. 73–87.

Vanderbei, R. S., M. S. Meketon and B. A. Freedman (1986), "A Modification of Karmarkar's Linear Programming Algorithm." Algorithmica, **1**, pp. 395–407.

Vanderbei, R. J. (1990), "ALPO: Another Linear Program Optimizer", Technical Report, AT&T Laboratories, Murray Hill, NJ.

Vanderbei, R. S. (1994), "Interior-Point Methods: Algorithms and Formulations." ORSA Journal on Computing, **6**(1), pp. 32–34.

Varga, R. S. (1962), Matrix Iterative Analysis, Prentice-Hall, Englewood Cliffs, NJ.

Vladimirou, H. (1990), "Stochastic Networks: Solution Methods and Applications in Financial Planning", Ph.D. Thesis, Princeton University, Department of Civil Engineering and Operations Research, Priceton, NJ.

Von Hohenbalken, B. (1977), "Simplicial Decomposition in Non-linear Programming Algorithms ." Mathematical Programming, **13**, pp. 49–68.

Wright, S. J. (1990), "Implementing Proximal Point Methods for Linear Programming." Journal of Optimization Theory and Applications, **65**(3), pp. 531–554.

Yamnitsky, B. (1982), "Notes on Linear Programming", MSc Thesis, Boston University.

Ye, Y. and M. Kojima (1987), "Recovering Optimal Dual Solutions in Karmarkar's Polynomial Algorithm for Linear Programming." Mathematical Programming, **39**(3), pp. 305–317.

Ye, Y., R. A. Tapia and Y. Zhang (1991), "A Superlinearly Convergent $O(\sqrt{n}L)$-Iterations Algorithm for Linear Programming", TR91-22, Department of Mathematical Science, Rice University, Houston, Texas.

Ye, Y., O. Guler, R. A. Tapia and Y. Zhang (1993), "A Quadratic Convergence $O(\sqrt{n}L)$-Iteration Algorithm for Linear Programming." Mathematical Programming, **59**, pp. 151–161.

Zenios, S. A., M. C. Pinar and R. S. Dembo, (1990), "A Smooth Penalty Function Algorithm for Network Structured Problems", Report 90-12-05, Decision Sciences Department, The Wharton School, University of Pennsylvania.

Zenios, S. A. and M. C. Pinar, (1992), "Parallel Block Partitioning of Truncated Newton for Non-linear Network Optimization." SIAM Journal on Scientific and Statistical Computing, **13**, (to appear).

Zhang, Y. and R. A. Tapia (1992), "Superlinear and quadratic Convergence of Primal-Dual Interior-Point Methods for Linear Programming Revisited." Journal of Optimization Theory and Applications, **73**(2), pp. 229–242.

Appendix A: Cholesky Decomposition

The Cholesky method is a variant of Gauss elimination for symmetric positive semidefinite n×n-matrix. If **M** is such a matrix it can be written in the factored form $\mathbf{M} = \mathbf{L}\mathbf{L}^T$, as depicted below.

$$\begin{pmatrix} M_{11} & M_{12} & \dots & M_{1n} \\ M_{21} & M_{22} & \dots & \\ \vdots & & & \\ M_{n1} & M_{n2} & \dots & M_{nn} \end{pmatrix} = \begin{pmatrix} L_{11} & & & 0 \\ L_{21} & L_{22} & & \\ \vdots & & & \\ L_{n1} & L_{n2} & \dots & L_{nn} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & \dots & L_{n1} \\ & L_{22} & \dots & L_{n2} \\ & & & \vdots \\ 0 & & & L_{n\,n} \end{pmatrix}$$

L is lower triangular and sometimes called the square root of **M**, given its similarity with the scale case. The method, due to Cholesky and Banachiewicz, is described in the following algorithm for computing **L**:

for i = 1, 2,..., n

$$L_{ii} = \sqrt{M_{ii} - \sum_{j=1}^{i-1} L^2_{ji}}$$

$$L_{ji} = \begin{cases} 0 & \text{for } j < i \\ \dfrac{1}{L_{ii}} \left(M_{ji} - \sum_{k=1}^{i-1} L_{jk} L_{ik} \right) & \text{for } j = i+1, \dots, n \end{cases}$$

Appendix B: Incomplete Cholesky Preconditioners

One of the most important preconditioning strategies used with conjugate gradient methods for solving the system of equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is symmetric PD, involves computing an incomplete Cholesky factorization of $\mathbf{A}$. The idea behind this approach is to calculate a lower triangular matrix $\mathbf{H}$ with the property that $\mathbf{H}$ has some tractable sparsity structure and is somehow "close" to $\mathbf{A}$'s exact Cholesky factor $\mathbf{G}$. The preconditioner is taken to be $\mathbf{M} = \mathbf{HH}^T$. To appreciate this choice consider the following facts:

- There exists a unique symmetric positive definite matrix $\mathbf{C}$ such that $\mathbf{M} = \mathbf{C}^2$.
- There exists an orthogonal $\mathbf{Q}$ such that $\mathbf{C} = \mathbf{QH}^T$, i.e., $\mathbf{H}^T$ is the upper triangular factor of a QR factorization of $\mathbf{C}$.

The heuristic is, therefore, obtained:

$$\mathbf{\check{A}} = \mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1} = \mathbf{C}^{-T}\mathbf{A}\mathbf{C}^{-1} = (\mathbf{HQ}^T)^{-1}\mathbf{A}(\mathbf{HQ}^T)^{-1}$$
$$\mathbf{Q}(\mathbf{H}^{-1}\mathbf{G}\mathbf{G}^T\mathbf{H}^{-T})\mathbf{Q}^T = \mathbf{I}$$

Thus, the better $\mathbf{H}$ approximates $\mathbf{G}$ the smaller the condition of $\mathbf{\check{A}}$, and the better the performance of the PCCGM.

An easy but effective way to determine such a simple $\mathbf{H}$ that approximates $\mathbf{G}$ is to step through the Cholesky reduction setting h_{ij} to zero if the corresponding a_{ij} is zero. Pursuing this with the outer product version of Cholesky obtains, [Golub & van Loan, (1983)]:

Step 1.**for** k = 1:n

$$A_{kk} = \sqrt{A_{kk}}$$

Step 2. **for** i = k+1:n

if $A_{ik} \neq 0$

$$A_{ik} = \frac{A_{ik}}{A_{kk}}$$

end

Step 3. **for** j = k+1:n

Step 4. **for** i = j:n

if $A_{ij} \neq 0$

$$A_{ij} = Aij - A_{ik}A_{jk}$$

end

Another version of the Cholesky factor [Setiono, (1990)] can be obtained by the procedure:

Step 1.**for** i = 1,2,...m

$$M_{ii} = \sqrt{(M_{ii} - \sum_{i-1}^{k=1} M^2_{ik})}$$

Step 2. **for** j = i+1,...m

if $M_{ji} \neq 0$

$$M_{ji} = \frac{M_{ji} - \sum_{i-1}^{k=1} M_{jk} M_{ik}}{M_{ii}}$$

end

Appendix C: The Variant of the Dual Projective of Karmarkar and Ramakrishnan

The two phases of the variant of the dual projective algorithm, [Karmarkar & Ramakrishnan, (1991)], as described in Chapter 4 is given below.

Algorithm A (Variant of the dual projective algorithm)

Step 1. Given an initial interior starting point $\mathbf{y^0}$, $\mathbf{s_1^0}$, $\mathbf{s_2^0}$, and parameters α and ε, α being the step-size parameter ($\alpha < 1$), and ε being the switching criterion, set (a) the iteration counter i = 0; (b) switch_criterion = false.

Step 2. **While** switch_criterion = false

Step 3. Objective function improving step.

(a)Compute ascent directions Δy, Δs_1 and Δs_2 using equations

$$\Delta y=(AD^2A^T)^{-1}(b-AD_1^2D_e^2u)$$
$$\Delta s_1=-D_e^2D_2^2A^T(AD^2A^T)^{-1}(b-AD_1^2D_e^2u)-D_e^2u$$
$$\Delta s_2=D_e^2D_1^2A^T(AD^2A^T)^{-1}(b-AD_1^2D_e^2u)-D_e^2u$$

where $D_e^{-2} = D_1^2 + D_1^2$, $D^2 = D_1^2 - D_1^2D_e^2D_1^2$,

$D = \begin{bmatrix} D_1 \\ & D_2 \end{bmatrix}$, $D_1^{-1} = \mathrm{diag}(s1)$, $D_2^{-1} = \mathrm{diag}(s_2)$.

(b)Compute maximum step-length permissible to keep s_1 and s_2 non-negative.

$$\beta = \min\left(\min_{\Delta s1k<0} -\frac{s_{1k}{}^i}{\Delta s_{1k}}, \min_{\Delta s2m<0} -\frac{s_{2m}{}^i}{\Delta s_{2m}}\right)$$

(c)Compute a fraction of the maximum step-length.

$$\beta = \alpha\beta.$$

(d)Update the solution.

$$y^{i+1} = y^i + \beta\Delta y,$$
$$s_1{}^{i+1} = s_1{}^i + \beta\Delta s_1,$$
$$s_2{}^{i+1} = s_2{}^i + \beta\Delta s_2,$$
$$i = i + 1.$$

Step 4. (a)Check if related improvement in objective function is less than ε

if

$((\mathbf{b}^T\mathbf{y}^i - \mathbf{u}^T\mathbf{s}_2^i) - (\mathbf{b}^T\mathbf{y}^{i-1} - \mathbf{u}^T\mathbf{s}_2^{i-1}) < \varepsilon |(\mathbf{b}^T\mathbf{y}^{i-1} - \mathbf{u}^T\mathbf{s}_2^{i-1})|$

then

swithch_criterion = true

end

Step 5. Reciprocal estimate step.

(a) Call Algorithm B to apply the reciprocal estimates to the current iteration and obtain $\Delta\mathbf{y}$, $\Delta\mathbf{s}_1$ and $\Delta\mathbf{s}_2$.

(b) Re-execute Steps 3(b) to 3(d) to update the solution.

end

Algorithm B (The reciprocal estimates algorithm)

Step 1. Given parameters ξ, ψ, and a feasible dual solution $(\mathbf{y}^*, \mathbf{s}_1^*, \mathbf{s}_2^*)$, estimate the initial primal variable $\mathbf{x}^0$, as follows:

(a)Compose the vectors $\bar{\mathbf{x}}$ and $\mathbf{x}'$.

Let $\mathbf{x}^0 = \bar{\mathbf{x}} + \sigma\mathbf{x}'$ where

$$\bar{x}_i = \begin{cases} 0 & \text{if } s_{2i}^* < s_{1i}^* \\ u_i & \text{if } s_{2i}^* \geq s_{1i}^* \end{cases}$$

and

$$x_i' = \begin{cases} \dfrac{1}{s_{1i}^*} & \text{if } s_{2i}^* < s_{1i}^* \\ \dfrac{-1}{s_{2i}^*} & \text{if } s_{2i}^* \geq s_{1i}^* \end{cases}$$

(b)Choose σ so that $\mathbf{Ax}^0$ is as close to $\mathbf{b}$ as possible. Thus

$$\sigma = \frac{(\mathbf{b} - \mathbf{A}\bar{\mathbf{x}})^T(\mathbf{Ax}')}{\|\mathbf{Ax}'\|_2^2}$$

(c)Perform the minimum ratio test to make $\mathbf{x}^0$ satisfy the bounds.

Compute

$$d = \min\left[\min_{1\leq i\leq n, s_{2i}^* < s_{1i}^*}(u_i s_{1i}^*), \min_{1\leq i\leq n, s_{2i}^* > s_{1i}^*}(u_i s_{2i}^*)\right]$$

Step 2. (a) stop_criterion = false; (b) k = 0.

Step 3. **While** (stop_criterion = false)

(a) $\mathbf{D} = \text{diag}(x_i^k)$.

(b) $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}^k$

(c) Solve the system $\mathbf{A}\mathbf{D}^2\mathbf{A}^T\Delta\mathbf{x} = \mathbf{r}$

(d) If in dual phase, interpret $\Delta\mathbf{x}$ as a direction in dual space, i.e., $\Delta\mathbf{y}$. Compute the corresponding directions for $\Delta\mathbf{s_1}$ and $\Delta\mathbf{s_2}$ and return to algorithm A.

(e) $\mathbf{x}^{k+1} = \mathbf{x}^k + \eta\mathbf{D}^2\mathbf{A}^T\Delta\mathbf{x}$, where

$$\eta = \psi^*\left(\min_{i,(D^2A^T\Delta x)_i>0} \frac{(u_i - x_i^k)}{(\mathbf{D}^2\mathbf{A}^T\Delta\mathbf{x})_i}, \min_{i,(D^2A^T\Delta x)_i<0} \frac{-x_i^k}{(\mathbf{D}^2\mathbf{A}^T\Delta\mathbf{x})_i}\right)$$

(f) **if** $(\|\mathbf{A}\mathbf{x}^{k+1}\,\mathbf{1} - \mathbf{b}\|_2 \le \|\mathbf{b}\|\xi)$

stop_criterion = true

else

k = k + 1

end

end

Appendix D: Interior Point Methods MATLAB Codes

Barnes Algorithm

```
function [x,iter,time,matrix_dip]=barnes(A,b,c,tol)
%Barnes Affine Transform Method {Math. Prog. 36(1986) 174-182}
%
%Initialization
%
t1=clock;x2=[];x=[];[m n]=size(A);AOR=A;matrix_dip=[];
%
%Set Up Initial Problem
%
aplus1=b-sum(A(1:m,:)')';cplus1=1000000;A=[A aplus1];
c=[c cplus1];B=[];n=n+1;x0=sparse(ones(1,n)');x=x0;
%
%Set alpha Barnes used 0.1
%
alpha = .0001;lambda=sparse(zeros(1,m)');
%
%Main Algorithm
%
iter=0;lambda'*b;
while abs(c*x-lambda'*b) >tol
        %disp('new one');
        lambda'*b;x2=x.*x;D=diag(x);D2=diag(x2);AD2=A*D2;
```

```
        lambda=(AD2*A')\(AD2*c');
        dualres=c'-A'*lambda;normres=norm(D*dualres);
        for i=1:n
                if dualres(i)>0
                                ratio(i)=normres/(x(i)*(c(i)-A(:,i)'*lambda));
                else
                                ratio(i)=inf;
                end
        end
        R=min(ratio)-alpha;x1=x-R*D2*dualres/normres;
        x=x1;basiscount=0;B=[];basic=[];cb=[];
        for k=1:n
                if x(k)>tol
                        basiscount=basiscount+1;
                        basic=[basic k];
                end
        end
%
%Only used if problem non-degenerate
%
        if basiscount<=m
                for k=basic
                                B=[B A(:,k)];cb=[cb c(k)];
                end
                primalsol=sparse(full(b')/full(B'));len_primalsol=length(primalsol);
                        dualsol=sparse(full(cb)/full(B));sol=primalsol;break;
        end
        iter=iter+1;
        if rem(iter,1)==0
```

```
            matrix_dip=[matrix_dip;iter c*x];
        end
        matrix_dip=full(matrix_dip);
end;
time=etime(clock,t1);
```

Infeasible Method

```
function [x,its,time_inf,matrix_dip]=infmethod(a,b,c,tol)
%Andersen Infeasible Dual Affine Scaling Method, {Scandinavian Workshop in Linear
%Programming, 1993}
%
%Set initial values
t3=clock;matrix_dip=[]; [m n]=size(a);k=n-m; lambda=.99999;iteration=0;
y=norm(c)/norm(a'*b)*b; yr=2*norm(a'*y-c'); x=sparse(ones(1,n)');
Climit=norm(b,inf);
%
%Main Loop
%
while abs(c*x-y'*b) >tol
            iteration=iteration+1;
            if rem(iteration,1)==0
                matrix_dip=[matrix_dip;iteration c*x];
            end
            matrix_dip=full(matrix_dip);
            s=c'-a'*y+yr*ones(1,n)';
            deltayr=-Climit*yr*yr;
            sminus2=diag(sparse(ones(1,n))./(s.*s)');
```

```
            as=a*sminus2;
            deldiv=(b+deltayr*as*sparse(ones(1,n)'));
            deltay=(as*a')\deldiv;
            %primal variables
            xval=a'*deltay-deltayr*sparse(ones(1,n)');
            x=sminus2*xval;
            deltas=-xval;

if(deltas>=0 & deltas~=0),disp('unbounded');break;end;
            minratio=[];
            for i=1:n
                        if deltas(i)<0
                                    minratio(i)=-s(i)/deltas(i);
                        else
                                    minratio(i)=inf;
                        end
            end;
            alpha=min(minratio);
            y=y+lambda*alpha*deltay;
            yr=yr+lambda*alpha*deltayr;
            if yr<=0,yr=0;end;
end
its=iteration;
t4=clock;
time_inf=etime(t4,t3);
```

Karmarkar and Ramakrishnan Dual Variant

```
function [primalsol,ITER_ALG_A,ITER_ALG_B,TIME1,TIME2,TIME,matrix_dip]=
karmar93(A,b,c,u,epsdual,epsprimal,nu,theta)
%
%This Program Is An Implementation Of A Variant Of Karmarkar's 1992 Projective
%Algorithm For Linear Programming
%
%take nu and theta as 1e-8 and epsdual as .005 and epsprimal .005
%
D1=[]; D2=[]; De2=[]; DDASH2=[]; y=[]; ya=[]; s1=[]; s1i=[]; s1a=[]; s1ai=[]; s2=[];
s2i=[]; s2a=[]; s2ai=[]; alfa=[]; alfai=[]; beta=[]; min1=[]; min1i=[]; min2=[]; min2i=[];
min3i=[]; min4i=[]; xdash=[]; xtilda=[];
%
%Finding Initial Solution.
%
TIME_BEGIN1=clock;[m,n]=size(A);ya=sparse(zeros(m,1));
for i=1:n
        vec(i)=norm(A(:,i));
end
alphai=abs(c./vec);alpha=max(alphai);s2a=2*alpha*vec';s1a=s2a+c';
%
%Dual Projective Algorithm.
%
ITER_ALG_A=0;super_counter=0;switch_criterion=1;
while switch_criterion==1
ITER_ALG_A=ITER_ALG_A+1;
%
%Step 3(A) Compute Ascent Directions Dy,Ds1,Ds2.
```

```
%
clear Dee2 DDASH2 Dy Ds1 Ds2
        s1aa=sparse(ones(n,1))./s1a;s2aa=sparse(ones(n,1))./s2a;D1=diag(s1aa);D2=diag(s
        2aa);Dee2=[];Dee2=sparse(ones(n,1))./(s1aa.^2+s2aa.^2);De2=diag(Dee2);D1squar
        e=D1^2;D2square=D2^2;DDASH2=D1square-(D1square*De2*D1square);
        Dy=(speye(size(A*DDASH2*A'))\(A*DDASH2*A'))*(b-A*D1square*De2*u);
        Ds1=-De2*D2square*A'*Dy-De2*u;Ds2=De2*D1square*A'*Dy-De2*u;
%
%(B) Compute Maximum Step Length Permissible To Keep S1 And S2 Non-Negative.
%
        beta=feastep(Ds1,Ds2,n,s1a,s2a);y=ya;s1=s1a;s2=s2a;
%
%(d) Update The Solution.
%
        ya=ya+beta*Dy;s1a=s1a+beta*Ds1;s2a=s2a+beta*Ds2;clear Dy Ds1 Ds2
%
%Step 4 Check If Relative Improvement In Objective Function Is Less Than Epsilon.
%
        if (b'*ya-u'*s2a)-(b'*y-u'*s2)<epsdual*abs(b'*y-u'*s2)
        switch_criterion=0;
        end
        xr=recipest(s1a,s2a,A,b,n,u);
%
%Step 2.
%
        D=diag(xr);r=b-A*xr;ada=A*D^2*A';Dy=solver(ada,r,m,n,theta,nu);test=norm(
        ada*Dy-r);Ds1=-De2*D2square*A'*Dy-De2*u;Ds2=De2*D1square*A'*Dy-De2*u;
%
%Re-execute Steps 3(B) To 3(D) To Update The Solution.
```

```
%
        beta=feastep(Ds1,Ds2,n,s1a,s2a);
%
%(d) Update The Solution.
%
        ya=ya+beta*Dy;s1a=s1a+beta*Ds1;s2a=s2a+beta*Ds2;objectvalue_dual=b'*ya;obje
        ctvalue_dual_pr=objectvalue_dual;
        if abs(objectvalue_dual-objectvalue_dual_pr)>1000*abs(objectvalue_dual_pr)
                disp('THE PROBLEM IS INFEASIBLE!')
                 disp('PROGRAM TERMINATED!')
                error('');
        end
end
TIME_END1=clock;TIME1=etime(TIME_END1,TIME_BEGIN1);
save data.dat ya s1a s2a ITER_ALG_A objectvalue_dual;
disp('DUAL PHASE IS COMPLETED, RESULTS ARE SAVED IN FILE DATA.DAT.')
disp('PLEASE WAIT,PROCEED IN PRIMAL PHASE.')
TIME_BEGIN2=clock;
%
%Calculate Primal Approximate Solution Using Reciprocals Of Slacks
%
xr=recipest(s1a,s2a,A,b,n,u);stop_criterion=1;
%
%This lOop Refines Primal Solution
%
ITER_ALG_B=0;
while (stop_criterion==1)
        ITER_ALG_B=ITER_ALG_B+1;
        modstep=norm(A*xr-b);
```

```
        D=diag(xr);r=b-A*xr;ada=D^2*A'; ada1=A*ada;
        xdash2=solver(ada1,r,m,n,theta,nu); primedir=ada*xdash2;
        for i=1:n
                if abs(primedir(i))<1e-10, primedir(i)=0;end;
        end
        for i=1:n
                if (primedir(i)>0),
                        minstep(i)=(u(i)-xr(i))/primedir(i);
        elseif (primedir(i)<0),
                minstep(i)=-xr(i)/primedir(i);
        else
                minstep(i)=Inf;
        end;
end;
stepval=min(minstep);
if modstep>.1, modstep=.05;end;
xstep=modstep*stepval; xr=xr+xstep*primedir; modstep=norm(A*xr-b);
objectvalue_primal=c*xr;
if rem(ITER_ALG_B,2)==0
        matrix_dip=[matrix_dip;ITER_ALG_B objectvalue_primal];
end
matrix_dip=full(matrix_dip);
if (abs(objectvalue_primal-objectvalue_dual)<0.0000005 & ITER_ALG_B>25)
        stop_criterion=0;
end
flops;aver_flops_per_iteration=flops/(ITER_ALG_A+ITER_ALG_B)
disp('PRIMAL PHASE IS COMPLETED, RESULTS ARE SAVED IN FILE
DATA.DAT.')
TIME_END2=clock;TIME2=etime(TIME_END2,TIME_BEGIN2);TIME=TIME1+TIME2;
```

```
primalsol=xr';
dualsol=ya';
save data.dat xr ITER_ALG_B objectvalue_primal TIME ;

%Functions Feastep, Solver And Recipest Are Called Within Karmar93

function v=feastep(Ds1,Ds2,n,s1a,s2a)
for k=1:n
        if(Ds1(k)<0)
                min1i(k)=-s1a(k)/Ds1(k);
        else
                min1i(k)=Inf;
        end
        if(Ds2(k)<0)
                min2i(k)=-s2a(k)/Ds2(k);
        else
                min2i(k)=Inf;
        end
end
betaval=min([min1i min2i]);
%
%(c) Compute A Fraction Of The Maximum Step Length.
%
v=.9*betaval;


function xr=recipest(s1a,s2a,A,b,n,u)
%uses reciprocals of slack variables to estimat primal solution
%
```

```
xtilda=sparse(ones(n,1));xdash=sparse(ones(n,1));
for i=1:n
        if(s2a(i)<s1a(i))
                xtilda(i)=0;
                xdash(i)=1/s1a(i);
        else
                xtilda(i)=u(i);
                xdash(i)=-1/s2a(i);
        end
end
%
%(b) Choose Sigma So That A*Xr Is As Close To B As Possible.
%
axd=A*xdash;sigma=(b-A*xtilda)'*axd/(norm(axd)*norm(axd));
%
%(c) Perform The Minimum Ratio Test To Make Xr Satisfy The Bounds.
%
for i=1:n
        if(s2a(i)<s1a(i))
                min3i(i)=u(i)*s1a(i);
        else
                min3i(i)=Inf;
        end
        if(s2a(i)>=s1a(i))
                min4i(i)=u(i)*s2a(i);
        else
                min4i(i)=Inf;
        end
 end
```

```
delta=min([min3i min4i]);
%
%(d) Check For Feasibility.
%
if sigma>= delta,
        sigma=0.9*delta;
end;
xr=xtilda+sigma*xdash;



function xdash=solver(ada,b,m,n,TITA,EPSILON_C)
%Given Parameters Ldash,Gdash,Tita,Epsilon,Mxitr And A System Of Equations
A*D^2*A'*X=B,Computes A Feasible X.
%
xdash=[]; gdash=[]; ddash=[]; qdash=[]; q=[];
mxitr=n;
%
%Step 1. Initialization.
%
xdash=sparse(zeros(m,1));
gdash=-b;
ddash=-gdash;
miouinit=b'*b;
if miouinit<1e-16,
        disp('zero b squared');
end;
stop_criterion1=1;
k=0;
miou=miouinit;
```

```
%
%Step 2.
%
while (stop_criterion1==1)
        qdash=ada*ddash;
        q=qdash;
        r=ddash'*q;
        if (r==0),
                error('r=0,divide by 0!!!')
        end
        alpha=miou/r;
        xdash=xdash+alpha*ddash;
        gdash=gdash+alpha*q;
        delta=gdash'*qdash;
        beta=delta/r;
        ddash=-gdash+beta*ddash;
        miou=beta*miou;
        k=k+1;
        val=ada*xdash;
        if((1-val'*b/(norm(val)*norm(b)))<=TITA)&(miou/miouinit<=EPSILON_C),
                stop_criterion1=0;
        end
        if(k>mxitr)
                stop_criterion1=0;
        end
end
```

Predictor Corrector Method

```
function [sol,iter,time,matrix_dip]=pdcorpred(a,b,c,tol)
%Mehrotra's method, Siam J. of Optimization 2(4), 575-601

% Set Up Initial Feasible Problem
%
t1=clock; x2=[]; x=[]; matrix_dip=[]; [m,n]=size(a);
%
% Set Up Initial Problem
%
[m,n]=size(a);
y0=sparse(ones(m,1));z0=sparse(ones(n,1));
aplus1=b-sum(a(1:m,:)')';
cplus1=10^6;ba=10^6;
newrow=(a'*y0+z0-c)';
a=[a aplus1 sparse(zeros(m,1))];
a=[a;newrow 0 1];
c=[c;cplus1];
c=[c;0];
n=n+2;m=m+1;
b=[b;ba];
x0=sparse(ones(1,n))';x=x0;x(n)=ba-newrow*x(1:n-2);
ya=-1;zb=1;y=[y0;ya];z=[z0;10^6-aplus1'*y0;zb];
%
%Set rho
rho=0.99995;
%
%Set Initial Y And Z And Mu
```

```
%
mu=0;
%
%Main Algorithm
%
iter=0;pd=1+tol;
while abs(pd)>tol
        %Set Up First Order Conditions
        bigz=diag(z);bigx=diag(x);
        %The lhs Matrix
        firstoa=[bigz sparse(zeros(size(a'))) bigx;a sparse(zeros(m)) sparse(zeros(m,n));
        sparse(zeros(n)) a' speye(n)];
        %The rhs of System
        firstob=[-bigx*bigz*sparse(ones(n,1));b-a*x;c-a'*y-z];
        %Now Solve System
        soldxyz=firstoa\firstob;
        %Calculate Correction
        dbarx=soldxyz(1:n);dbary=soldxyz(n+1:m+n);dbarz=soldxyz(m+n+1:2*n+m);
        dx=dbarx;dz=dbarz;
        for i=1:1
                cor=diag(dx)*diag(dz)*sparse(ones(n,1));
                rirstob=firstob-[-mu*sparse(ones(n,1))+cor;sparse(zeros(m+n,1))];
                soldxyz=firstoa\firstob;
                dx=soldxyz(1:n);dy=soldxyz(n+1:n+m);dz=soldxyz(n+m+1:2*n+m);
        end
        %Do Only One Correction
        %Calculate New x,y,z By Taking Feasible Step
        ratio=[];
        for i=1:n
```

```
            if dx(i)<0
                ratio(i)=-x(i)/dx(i);
            else
                ratio(i)=inf;
            end
        end
        alphap=min([min(ratio),1]);
        ratio=[];
        for i=1:n
            if dz(i)<0
                ratio(i)=-z(i)/dz(i);
            else
                ratio(i)=inf;
            end
        end
        alphad=min([min(ratio),1]);
        x1=x+rho*alphap*dx;xbar1=x+alphap*dbarx;
        y1=y+rho*alphad*dy;ybar1=y+alphad*dbary;
        z1=z+rho*alphad*dz;zbar1=z+alphad*dbarz;
        g=xbar1(1:n-2)'*zbar1(1:n-2);
        mu=(g/(x(1:n-2)'*z(1:n-2)))^2*g/(n-2);iter=iter+1;
        pd=c(1:n-2)'*x1(1:n-2)-y1(1:m-1)'*b(1:m-1);%full([mu pd])
        x=x1;y=y1;z=z1;
        if rem(iter,1)==0
            matrix_dip=[matrix_dip;iter c'*x];
        end
        matrix_dip=full(matrix_dip);
end
sol=x(1:n-2);time=etime(clock,t1);
```

Appendix E: Quadratic Programming Algorithms MATLAB Codes

GL Interior Point Method

```
%The Following Coding Is A Modification Of Burrett (1994) Code. Different Convergence
%Criterion and Formula For Updating The Penalty Parameter Is Used And Sparsity Is
%Taken Into Account
function [sol,iters]=goldlu(Q,p,d,f)
%Quadratic Solver Taken From:D. Goldfarb & S. Lui Mathematical Programming 49 (1991)
%pp:325-340
%
%Sparsity Is Obtained In The Generation Of The Quadratic Problems. The MATLAM Code
%of the Generators Can Be Found in Appendix F.
%Step 1. Initialization
%
[m,n]=size(d);m1=2^8;tau=0.0005;eps0=2^8;rho=2^8; gamma=0; sigma=0.9; iters=0;
oldmu=0.5;epsval=eps0;dualgap=1;
%
%Step 2. Formulate Initial Problem
%Step 2 Was Not Used In The Experiments Performed To Obtain The Results In Chapter 5.
%But It Is Described Here For The Sake Of Completeness
%
g=[Q zeros(size(Q),1) zeros(size(Q),1);zeros(1,size(Q)) 0 0;zeros(1,size(Q)) 0 0];
a=[d f/rho-d*ones(1,n)' zeros(size(d),1);ones(1,n) 1 1];
c=[p' m1 0];d=[f/rho;n];n=n+2;x=ones(n,1);z=0.001*ones(n,1);
```

```
%Step 3. Main Loop
%
while dualgap>tau
        iters=iters+1;if rem(iters,20)==0,disp(iters),end
        if abs(epsval-oldmu)/oldmu>gamma
                munew=epsval;
        else
                munew=oldmu;
        end
        oldmu=munew;znew=[];
        for i=1:n
                if abs(x(i)-z(i))/z(i)>gamma
                        znew(i)=x(i);
                else
                        znew(i)=zold(i);
                end
        end
        zold=znew;smallg=g*x+c'-epsval*diag(ones(n,1)./x)*ones(n,1);
        dz=diag(ones(1,n)./znew.^2);ghat=g+munew*dz;
        ghatin=ghat\eye(n);y=(a*ghatin*a')\a*ghatin*smallg;
        h=ghatin*(a'*y-smallg);x=x+h;s=g*x+c'-a'*y;
        pv=p*x +0.5*x'*Q*x;dv=f*y-0.5*x'*Q*x;
        %dualgap=x'*s;
        %Originally Proposed Termination Criterion
        dualgap=abs((pv-dv)/pv);
        epsval=(pv-dv)/n^2;
        %epsval=(1-sigma/sqrt(n))*epsval;
        %Originally Proposed Function For The Reduction Of epsval
end,sol=rho*x(1:n-2)';
```

CG Interior Point Method

```
%The Following Coding Is A Modification Of Burrett (1994) Code. In This Implementation
%Sparsity Is Taken Into Account

function [x,loops,it,matrix_dip]=cg_quad_solver_s(A,b,c,Q)
%Quadratic Solver Using Conjugate Gradients Computational Optimization And
Applications, 2, 5-28 (1993)

%Step 0. Initialization
%
[m,n]=size(A);z=(A*A')\speye(m);p=speye(n)-A'*z*A;matrix_dip=[];
x=sparse(ones(n,1));it=0;e=x;mu=norm(z*A*c')/norm(z*A*e);t1=clock;
optimality_criterion=0;stoping_criterion=0;loops=0;pv=2;mat1=[];mat2=[];
%
%Main Loop
%
while optimality_criterion==0
        k=0;loops=loops+1;y=diag(sparse(ones(n,1))./x);
        if rem(loops,20)==0,disp(loops),end
        grad=c'+Q*x-mu*y*e;g=grad;dx=sparse(zeros(n,1));
        d=-p*g;y2=diag(sparse(ones(n,1))./x.^2);h=Q+mu*y2;
        while norm(p*g)>0.0005
                k=k+1;dhd=d'*h*d;gamma=-d'*g/dhd;dx=dx+gamma*d;
                g=h*dx+grad;beta=d'*h*p*g/dhd;d=-p*g+beta*d;it=it+1;
        end
%
%Step 2. Force Feasibility
%
```

```
        ratios=[];
        for i=1:n
                if dx(i)<0
                        ratios=[ratios -x(i)/dx(i)];
                else
                        ratios=[ratios inf];
                end
        end
        alphav=min(ratios)*0.995;x=x+alphav*dx;
%
%Step 3. Compute Dual Variables
%
        y=(A*diag(x)*diag(x)*A')\(A*diag(x)*diag(x)*(c'+Q*x));
%
%Calculate Primal And Dual Objective Values & Check For Optimality
%
        pv_pr=pv;mu_pr=mu;dv=b'*y-0.5*x'*Q*x;pv=c*x+0.5*x'*Q*x;
        mu=max(10^(-9),mu/2);
        if rem(loops,2)==0
                matrix_dip=[matrix_dip;loops pv];
                mat1=[mat1;etime(clock,t1)];
                mat2=[mat2;flops];
        end
        matrix_dip=full(matrix_dip);
        if abs((pv-dv)/pv)<0.0005
                optimality_criterion=1;x=x';
        end
end
mat1,mat2
```

UDA Interior Point Method

```
function [x_sol,singl_qp_solver_counter,matrix_dip]=qp_solver(p,D,A,b,l,u)
%This Function Is Used To Find The Solution To A Quadratic Problem Of The Form
%min (p'*z + 1/2*z'*D*z), s.t.: A*z=b, l<z<u By Given Vectors p, l, u, b, And
%Matrices D And A, Where D Is Symmetric Positive Definite.

%Step 1. Initialization.
%
[m,n]=size(A);y=sparse(ones(m,1));iter_conj_grad=0;x=sparse(0.5*ones(n,1));
stop_rule=0;epsilon1=0.001;teta_sum=0;teta_count=0;singl_qp_solver_counter=0;
teta_count=0;teta_count_matrix=[];teta_matrix=[];teta_index=0;mat1=[];mat2=[];
epsilon2=0.0005;teta_temp=0;count_temp=0;matrix_dip=[];pv=100;t1=clock;
%
%Main Loop
%
while stop_rule==0
%STep 2. Computing Parameter Delta.
%
        Anadelta=b-A*x;
        if rem(iter_conj_grad,m)==0
                delta=Anadelta;
        else
                Anadelta_pr=b-A*x_pr;
                delta=Anadelta+(((Anadelta)'*(Anadelta-
Anadelta_pr))/(Anadelta_pr'*Anadelta_pr))*delta;
        end
        iter_conj_grad=iter_conj_grad+1;x_pr=x;
        if rem(iter_conj_grad,20)==0
```

```
            disp(iter_conj_grad)
        end
%
%Step 3. Solving The Subproblem And Computing Parameter Teta.
%
        [x,teta]=quad_presolver_s(diag(D)',(y'*A-p')',(delta'*A)',delta'*b,l,u);
        singl_qp_solver_counter=singl_qp_solver_counter+1;
        teta=-teta;
%
%Step 4. Update Parameter Y.
%
        y=y+teta*delta;
%
%Step 5. Optimality Check.
%
        pv_pre=pv;dv=b'*y-0.5*x*D*x';pv=p'*x'+0.5*x*D*x';
        if rem(iter_conj_grad,2)==0
            matrix_dip=[matrix_dip;iter_conj_grad pv];
            mat1=[mat1;etime(clock,t1)];
            mat2=[mat2;flops];
        end
        matrix_dip=full(matrix_dip);
        if abs(dv-pv)/pv<=epsilon2 | iter_conj_grad==300
            stop_rule=1;x_sol=x;
        else
            stop_rule=0;
        end
        x=x';
end %(WHILE)
```

```
% Functions Quad_Presolver, Ptsolve, Quadsolve2_s, Convertor, Setsub And Min_Eig Are
%Called Within Qp_solver


function [result,teta]=quad_presolver(d,a,b,b0,L,U)
%This Function Is Used To Check The Format Of A Singl Constraint Problem And Find
Any Solution Before The Main Procedure Is Called.

%Step 0. Initialization.
%
n=length(L);xsol=[];isol=[];resisol=[];inprop_criterion=0;temp_value=0;
aNEW=[];bNEW=[];dNEW=[];LNEW=[];UNEW=[];neg=[];fl=0;
%
%Step 1. Checking Vector D.
%
for i=1:n
        if d(i)<=0
                inprop_criterion=1;
        end
end
if inprop_criterion==1
        disp('THE SINGLY CONSTRAINT QUADRATIC PROBLEM IS NOT IN
        APPROPRIATE FORM.')
        error('ALL THE COEFFICIENTS OF THE SQUARES MUST BE POSSITIVE.')
end
%
%Step 2. Checking Bounds.
%
test_bounds=U-L;index=find(test_bounds<0);
```

```
if index~=[]
        disp('THE SINGLY CONSTRAINT QUADRATIC PROBLEM IS NOT IN
        APPROPRIATE FORM.')
        error('UPPER BOUNDS MUST BE GREATER THAN LOWER BOUNDS.')
end
%
%Step 3. Checking Vector B.
%
for i=1:n
        if b(i)==0 & -a(i)/d(i)>0 & -a(i)/d(i)>=L(i) & -a(i)/d(i)<=U(i)
                xsol=[xsol -a(i)/d(i)];isol=[isol i];
        elseif b(i)==0 & -a(i)/d(i)>0 & -a(i)/d(i)>U(i)
                xsol=[xsol U(i)];isol=[isol i];
        elseif b(i)==0 & -a(i)/d(i)>0 & -a(i)/d(i)<L(i)
                xsol=[xsol L(i)];isol=[isol i];
        elseif b(i)==0 & -a(i)/d(i)<0
                xsol=[xsol L(i)];isol=[isol i];
        elseif b(i)<0
                bNEW=[bNEW -b(i)];aNEW=[aNEW -a(i)]; temp_value=L(i);
                resisol=[resisol i]; LNEW=[LNEW -U(i)];UNEW=[UNEW -temp_value];
                dNEW=[dNEW d(i)];neg=[neg i];fl=1;
        else
                resisol=[resisol i];LNEW=[LNEW L(i)];UNEW=[UNEW U(i)];
                bNEW=[bNEW b(i)];dNEW=[dNEW d(i)];aNEW=[aNEW a(i)] ;fl=1;
        end
end
if fl==0
        result=xsol;
else
```

```
%
%Step 4. Solving The Reduced Quadratic Problem
%
        [teta,res]=quadsolve2(dNEW,aNEW',bNEW',b0,LNEW',UNEW');
%
%Step 5. Updating The Solution
%
        n1=length(isol);n2=length(resisol);n=n1+n2;n3=length(neg);
        for i=1:n
                for k=1:n2
                        if i==resisol(k)
                                result(i)=res(k);
                        end
                end
        end
        for i=1:n
                for j=1:n1
                        if i==isol(j)
                                result(i)=xsol(j);
                        end
                end
        end
        for i=1:n
                for k=1:n3
                        if i==neg(k)
                                result(i)=-result(i);
                        end
                end
        end
```

```
end

function xt=ptsolve(t,a,b,d,l,u)
%Solves Parametric Problem
n=length(a);
for i=1:n
        v(i)=(a(i)-b(i)*t)/d(i);
                if v(i)<=l(i)
                        xt(i)=l(i);
                elseif v(i)>= u(i)
                        xt(i)=u(i);
                else
                        xt(i)=v(i);
        end
end

function [topt, xt]=quadsolve2_s(d,a,b,b0,l,u)
% Initialization
%
il=[];iu=[];im=[];n=length(d);tl=[];tu=[];xt=[];x=[];iter=0;flag=0;
for i=1:n
        tl(i)=(a(i)-l(i)*d(i))/b(i);
        tu(i)=(a(i)-u(i)*d(i))/b(i);
end
%
% Bracketing
%
t1=min([tl tu]);tr=max([tl tu]);
xt1=ptsolve(t1,a,b,d,l,u);xtr=ptsolve(tr,a,b,d,l,u);
```

```
%
%  Checking Optimality and Infeasibility
%
if xt1*b==b0
        disp('THE SOLUTION IS OPTIMAL.')
        x=xt1,break;flag=1;
elseif xtr*b==b0
        disp('THE SOLUTION is OPTIMAL.')
        x=xtr,break
elseif xt1*b<b0 | xtr*b>b0
        disp('THE PROBLEM HAS NO FEASIBLE SOLUTION.'),break;flag=1;
else
        tmin=t1;tmax=tr;iset=1:n;
end
%
%  Main Loop
%
ilist=iset;
while length(iset)~=0
        tlm=median(tl(iset));
        isetgr=[];
        for i=iset
                if tl(i)>=tlm
                        isetgr=[isetgr i];
                end
        end
        tum=median(tu(isetgr));
        for t=[tlm tum]
                if t<tmax & t>tmin
```

```
                xt=ptsolve(t,a,b,d,l,u);
                if xt*b==b0
                        disp('THE SOLUTION IS OPTIMAL')
                        flag=1; x=xt;iset=[];
                elseif xt*b>b0
                        tmin=max(tmin,t);
                else
                        tmax=min(tmax,t);
                end
        end
    end
    ilset=sparse([]);iuset=sparse([]);imset=sparse([]);
    for j=iset
        if tl(j)<=tmin
                xt(j)=l(j);il=[il j];ilset=[ilset j];
        elseif tmax<=tu(j)
                xt(j)=u(j);iu=[iu j];iuset=[iuset j];
        elseif tmin<=tmax
                if tmax<=tl(j) & tu(j)<=tmin
                        xt(j)=(a(j)-b(j)*t)/d(j);
                        im=[im j];imset=[imset j];
                end
        end
    end
    iset=setsub(iset,iuset);
    iset=setsub(iset,ilset);
    iset=setsub(iset,imset);
end%(WHILE)
%
```

```
% Now Calculate
%
topt=sum(b(iu).*u(iu))+sum(b(il).*l(il))+sum(b(im).*a(im)./d(im)')-b0;
topt=topt/sum(b(im).^2./d(im)');
if im~=[]
        xt(im)=(a(im)-topt*b(im))'./d(im);
end
if il~=[]
        xt(il)=l(il);
end
if iu~=[]
        xt(iu)=u(iu);
end
xt=sparse(xt);topt=sparse(topt);



function [p,D,A,b,l,u]=convertor(p,D,A,b,l,u)
%This Function Is Used To Convert A Nonseparable Quadratic Problem To A Separable
One.

delta=min(eig(D));
%delta=min_eig(D);
bound=10000000;
if delta>0
        disp('MATRIX D IS POSITIVE DEFINITE.')
        if delta>0
                delta=0.9*delta;
        end
        mat=D-delta*eye(size(D));G=chol(mat);
```

```
        D_temp1=[delta*0.5*eye(size(D)) zeros(size(D))];
        D_temp2=[zeros(size(D)) 0.5*eye(size(D))];
        D=[D_temp1;D_temp2];
        b=[b;zeros(length(p),1)];
        p=[p;zeros(length(p),1)];
        A=[A zeros(size(A))];
        A_temp=[G -eye(size(G))];
        A=[A;A_temp];
        l=[l ; -bound*ones(size(l))];
        u=[u ; bound*ones(size(u))];
        disp('THE PROBLEM IS TRANSFORMED TO A SEPARABLE ONE.')
else
        disp('TRANSFORMATION IS NOT APPLICABLE FOR THIS PROBLEM.')
end

function v=setsub(s1,s2)
%Subtract Set S2 From Set S1
ls1=length(s1);ls2=length(s2);s1h=s1;
if ls2~=0 & ls1~=0
        for i=1:length(s1h)
                for j=1:length(s2)
                        if s1h(i)==s2(j)
                                s1h(i)=[];
                        end
                end
                if length(s1h)==ls1-ls2
                        break
                end
        end
```

```
else
        v=s1;
end
v=s1h;

function l=min_eig(a)
%This Function Calculates The Minimum Eigenvalue Of A Symmetric Or  Non Symmetric
%Real Matrix By The Inverse Iteration Algorithm. We Assume That Min_eig > -10

[m,n]=size(a);u=ones(n,1);termination_criterion=0;
loops=0;miou=-10;[L,U]=lu(a-miou*eye(n));
while termination_criterion==0
        loops=loops+1;u_pr=u;
        z=L\u;v=U\z;u=v/max(abs(v));
        if max(abs(u-u_pr))<10^(-50)
                        termination_criterion=1;
                        l=miou+1/max(abs(v));
        end
end
```

Appendix F: Decomposition Algorithms MATLAB Codes

Decentralized Algorithm

```
function [tt,time_dec,xx,main_loop,disp_matrix] = decomposition(AH, bH, cH,
lo_bounds_H, up_bounds_H,index_matrix)
%This Program Is An Implementation Of The Decentralized Decomposition Algorithm Of
%P.Mahey & P.D.Tao.

%Step 1. Initialization
%
UDASH=[];VDASH=[];optimality_criterion=0;[p,r]=size(index_matrix);[m,n]=size(AH);
epsilon=.01;lamda=1000;main_loop=1;x=[];b_Coupl_Sub_Matrix=[]; t1=clock; t6=0;
time_par=[];b_coupl_Sub_Matrix=[]; A_Coupl=AH(index_matrix(p,2)+1:m,:);
x_pr=sparse(ones(n,1)); disp_matrix=[]

%Initializing Matrix U
%
Ufirst=sparse(rand(m-index_matrix(p,2),1));
for i=1:p
        U=[U Ufirst];
end
A=U;
 %
%Initializing Matrix V
%
```

```
V=U(:,1:p-1);[mV,nV]=size(V);
if nV==1
        Vlast=-V;
else
        Vtemp=V';Vlast=(-sum(Vtemp))';
end
V=[V Vlast];B=V;
%
%Initializing Partitioning Of The RHS Corresponding To The Coupling Constraints
%
for i=1:p
        b_coupl_Sub_Matrix=[b_coupl_Sub_Matrix (sum((A_Coupl(:,index_matrix(i,3) :
        index_matrix(i,4)))'))'];
end
t2=etime(clock,t1);
%
%Main Loop
%
while optimality_criterion==0
%
%Step 2. Solve The Quadratic Subproblems And Compute U And V.
%
 for i=1:p
        t3=clock;index=index_matrix(i,:);Asub=AH(index(1):index(2),index(3):index(4));
        bsub=bH(index(1):index(2));csub=cH(index(3):index(4));usub=U(:,i);
        vsub=V(:,i);A_Coupl_Sub=A_Coupl(:,index(3):index(4));
        lo_bounds_sub=lo_bounds_H(index(3):index(4));
        up_bounds_sub=up_bounds_H(index(3):index(4));
        b_Coupl_Sub=b_coupl_Sub_Matrix(:,i);
```

```
%
%Computing The Parameters For The Quadratic Solver
%
        pp=csub'+(A_Coupl_Sub'*usub)+lamda*A_Coupl_Sub'*(vsub-b_Coupl_Sub);
        D=(lamda/2)*A_Coupl_Sub'*A_Coupl_Sub;
%
%Solving The Quadratic Subproblems
%
        [x_sol,singl_qp_solver_counter]=qp_solver(pp,D,Asub,bsub,lo_bounds_sub,
        up_bounds_sub);
        time_par=[time_par etime(clock,t3)];  t4=clock;x=[x x_sol];
%
%Computing U and V
%
        udash=usub+lamda*(A_Coupl_Sub*x_sol'+vsub-b_Coupl_Sub);
        UDASH=[UDASH udash]; vdash=b_Coupl_Sub-A_Coupl_Sub*x_sol';
        VDASH=[VDASH vdash];
 end
 %
%Step 3. Compute Projections
%
sum_proj=sum(VDASH');
for i=1:p
        sum_proj_matrix=[sum_proj_matrix sum_proj'];
end
V=VDASH-sum_proj_matrix/n;VDASH=[];
U=UDASH+sum_proj_matrix/n;UDASH=[];
t5=etime(clock,t4);
```

```
%Step 4. Check For Optimality
%
ax_b=[ax_b max(abs(AH*x'-bH))],res_1=[res_1 max(abs(ones(size(x'))-x'))]
 if max(abs(ones(size(x'))-x'))<=epsilon
        optimality_criterion=1;xx=x;
 else
        main_loop=main_loop+1;x_pr=x';x=[];pp=[];D=[];sum_proj_matrix=[];
        t6=t6+max(time_par)+t5;time_par=[];
end
end %(WHILE)
tt=etime(clock,t1);time_dec=t2+t6;
```

Augmented Lagrangian Method

```
function [timeII,time_parallel,xx,main_loop,disp_matrix]=decomposII(A,b,c,index_matrix)
%This Program Is An Implementation Of The Decomposition Algorithm Of J.M. Mulvey &
%A. Ruszcynski.

%Step 1. Initialisation
epsilon=1;termination_criterion=0;[m,n]=size(index_matrix);x=[];main_loop=0;test=[];
r=10^5;[mA,nA]=size(A);x_sol=[];t21=clock;time_dec_sub2=[];
pgr=sparse(ones(mA-index_matrix(m,2),1));sumQji=sparse(zeros(mA-
index_matrix(m,2),1)); t=0.95;xdash=sparse(ones(nA,1));u=2*sparse(ones(1,nA));
sumsub=sparse(zeros(size(pgr)));t27=0;stop_criterion=0;inner_loop=0;D=sparse(zeros(inde
x_matrix(1,4)-index_matrix(1,3)+1));disp_matrix=[];

%Main Loop
%
while termination_criterion==0
```

```
        main_loop=main_loop+1;
%
%Step 2. Solving The Subproblems
%
        while stop_criterion==0
                inner_loop=inner_loop+1;
                disp(full([main_loop inner_loop]))
                for i=1:m
                        index=index_matrix(i,:);t23=clock;
%
%Computing The Parameters For The Quadratic Solver
%
        Asub=A(index(1):index(2),index(3):index(4));bsub=b(index(1):index(2));
        csub=c(index(3):index(4));
        pp=c(index(3):index(4))-pgr'*A(index_matrix(m,2)+1: mA,index(3) :index(4));
                for j=1:m
                        if j~=i
sumQji=sumQji+A(index_matrix(m,2)+1: mA, index_matrix(j,3): index_matrix(j,4))
*xdash(index_matrix(j,3) :index_matrix(j,4));
                        end
                end
        Q=A(index_matrix(m,2)+1:mA,index(3):index(4));up_bounds_sub=u(index(3):inde
x(4));D=r*Q'*Q;
                        lo_bounds_sub=zeros(size(up_bounds_sub));
                        pp=[pp-r*((b(index_matrix(m,2)+1:mA)-sumQji)'*Q)]';
                        [mD,nD]=size(D);
                        for ii=1:mD
                                if D(ii,ii)==0
                                        D(ii,ii)=10^(-5);
```

```
                end
            end
        [x_sol,singl_qp_solver_counter]=qp_solver(pp,D,Asub,bsub,lo_bounds_sub,up_b
ounds_sub); x=[x;sparse(x_sol')];pp=[];D=[];sumQji=sparse(zeros(size(sumQji)));
t24=clock; t_dec22=etime(t24,t23);
            if main_loop==1
                time_dec_sub2=[time_dec_sub2 t_dec22];
            else
                time_dec_sub2(i)=time_dec_sub2(i)+t_dec22;
            end
        end
%
%Step 3. Checking For Optimality
%
        t25=clock;
        for i=1:m
            index=index_matrix(i,:);
        subtest=norm(A(index_matrix(m,2)+1:mA,index(3):index(4))*(x(index(3):index(4))
-xdash(index(3):index(4))));
            if subtest<=epsilon
                test=[test 1];
            else
                test=[test 0];
            end
        end

        if all(test)==1 | inner_loop==20
            stop_criterion=1;inner_loop=0;
            for i=1:m
```

```
                    index=index_matrix(i,:);

        sumsub=sumsub+A(index_matrix(m,2)+1:mA,index(3):index(4))*x(index(3):index(
4));
                    end

                else
                    xdash=xdash+t*(x-
xdash);x=[];test=[];sumsub=sparse(zeros(size(sumsub)));
                end
        end%(WHILE)
        stop_criterion=0;
        if max(abs(ones(size(x'))-x'))<=0.01
                termination_criterion=1;xx=x;
        elseif main_loop>8
                termination_criterion=1;x,xx=[];
        else
%
%STEP 4. UPDATING THE SOLUTION
%
                pgr=pgr+r*(b(index_matrix(m,2)+1:mA)-sumsub); x=[]; test=[];
                sumsub=sparse(zeros(size(sumsub)));%r=10*r;
        end
        t26=clock;t27=t27+etime(t26,t25);
end %(WHILE)
t22=clock;timeII=etime(t22,t21);time_parallel=max(time_dec_sub2)+t27;time_ratio2=timeII/
time_parallel;
```

Linear Quadratic Decomposition Algorithm With Penalty Function Φ_1

```
function [xx,main_loop,t_decIII,tIII_par,disp_matrix]=decomp_li(A,b,c,u,index_matrix)
%This Program Is An Implementation Of The Decomposition Algorithm Of M.C. Pinar &
%S.A. Zenios.

%Step 1. Initialization
%
[m,n]=size(index_matrix);x=[];termination_criterion=0;main_loop=0;[mA,nA]=size(A);
xi=[];s=mA-index_matrix(m,2);epsilon=.01;test1=[];test2=[];V=[];;D=zeros(nA,nA);
sf=zeros(size(c));t31=clock;t_matrix=[];M=10^(3);cho_int_alg=[];tol=0.0005;nta1=0.5;
nta2=0.5;nta3=0.5;disp_matrix=[];;t_s=0;t3t=0;
%
%Step 2. Solving The Relaxed Problem
%
t35=clock;
if A(index_matrix(m,2)+1:m,:)*x<=b(index_matrix(m,2)+1:m)
        termination_criterion=1;xx=x;
else
        x_pr=x;miou=max(abs(c));eps=max(0,nta3*max(A*x-b));
end,t36=etime(clock,t35);
%
%Main Loop
%
while termination_criterion==0
        t37=clock;main_loop=main_loop+1;
%Step 3. Computing The Parameters Of The Penalty Function
%
        E=A(index_matrix(m,2)+1:mA,1:nA);d=b(index_matrix(m,2)+1:mA);
```

```
        for j=1:s
                y=abs(E(j,:)*x-d(j));
                if y<=0.005
                        %test1=[test1 1];
                        disp('COUPLING CONSTRAINT SATISFIED.')
                elseif y>0.005 & y<=eps
                        disp('ADDING QUADRATIC SEGMENT.');
                        D=D+miou*(E(j,:)'*E(j,:))/2*eps;sf=sf-
miou*d(j)*E(j,:)/eps;test1=[test1 0];
                else
                        disp('ADDING LINEAR SEGMENT.');
                        sf=sf+miou*E(j,:);test1=[test1 1];V=[V;y];
                end
        end
        for iD=1:nA
                if D(iD,iD)==0
                        D(iD,iD)=10^(-5);
                end
        end,t38=etime(clock,t37);
%
%Step 4. Solving The Penalty Function
%
        ind=index_matrix(m,:);
        if all(test1)==1
                disp('LINEAR MASTER-PENALTY PROBLEM SOLVED BY
                PREDICTOR CORRECTOR ALGORITHM.');
                [x,iter]=pdcorpred(A(1:ind(2),1:nA),b(1:ind(2)),(c+sf)',tol);
        else
```

```
                disp('QUADRATIC MASTER-PENALTY PROBLEM.');
[x,singl_qp_solver_counter,ts]=simp_decomp(D,(c+sf),A(1:ind(2),1:nA),b(1:ind(2))
,index_matrix); t_s=t_s+ts;
        end
%
%Step 5. Checking For Optimality And Updating Parameters Miou And Eps
%
        t39=clock;
        if   max(abs(ones(size(x'))-x'))<=epsilon
                termination_criterion=1;xx=x;
        else
                if sum(test1)==0
                        eps=max(epsilon,nta1*eps);
                else
                        miou=miou*max(V)/(nta2*eps);
                end
                x_pr=x;test1=[];D=zeros(size(D));sf=zeros(size(sf));

        end
        t3t=t3t+t38+etime(clock,t39);disp_matrix=[disp_matrix;main_loop c*x];
end %(WHILE)
t32=clock;t_decIII=etime(t32,t31);tIII_par=t36+t_s+t3t;
```

Linear Quadratic Decomposition Algorithm With Penalty Function Φ_2

```
function [xx,main_loop,t_decIII,tIII_par,disp_matrix] = decomp_li_me (A,b,c,u,
index_matrix)
%This Program Is An Implementation Of A Variant Of The Decomposition Algorithm Of
%M.C. Pinar & S.A. Zenios.

%Step 1. Initialization
%
[m,n]=size(index_matrix);x=[];termination_criterion=0;main_loop=0;[mA,nA]=size(A);
xi=[];s=mA-index_matrix(m,2); epsilon=.01;test_1=[];test2=[];V=[]; D=zeros(nA,nA);
sf=zeros(size(c));t31=clock; t_matrix=[];M=10^(3) ;cho_int_alg=[]; tol=0.0005; nta1=0.5;
nta2=0.5;nta3=0.5;t_s=0;t3t=0;disp_matrix=[];
%
%Step 2. Solving The Relaxed Problem
%
for i=1:m
        t33=clock;
        [xi,iter]=pdcorpred(A(index(1):index(2),index(3):index(4)),b(index(1):index(2)),
        c(index(3):index(4))',tol);
        x=[x;xi];
        t34=clock;t_matrix=[t_matrix etime(t34,t33)];
end
t35=clock;
if max(abs(A(index_matrix(m,2)+1:m,:)*x-b(index_matrix(m,2)+1:m)))<epsilon
        termination_criterion=1;xx=x;
else
        x_pr=x;miou=max(abs(c));eps=nta3*max(A*x-b);
```

```
end,t36=etime(clock,t35);
%
%Main Loop
%
while termination_criterion==0
        t37=clock;main_loop=main_loop+1;
%
%Step 3. Computing The Parameters Of The Penalty Function
%
        E=A(index_matrix(m,2)+1:mA,1:nA);d=b(index_matrix(m,2)+1:mA);
        for j=1:s
                y=E(s,:)*x-d(s);
                if y<=0
                        test1=[test1 0];
                elseif y>=0 & y<=eps

        D=[D+(E(s,:)'*E(s,:))/eps];sf=[sf+2*d(s)*E(s,:)/eps+E(s,:)/2];test1=[test1 0];
                else
                        sf=[sf+(E(s,:)-eps/2)];test1=[test1 1];
                        V=[V;y];
                end
        end
        for iD=1:nA
                if D(iD,iD)==0
                        D(iD,iD)=10^(-5);
                end
        end,t38=etime(clock,t37);
%
%Step 4. Solving The Penalty Function
```

```
%
        ind=index_matrix(m,:);
        if all(test1)==1
                disp('ADDING LINEAR SEGMENT.');
                disp('MASTER-PENALTY PROBLEM SOLVED BY PREDICTOR
                CORRECTOR ALGORITHM.');
                [xi,iter]=pdcorpred(A(1:ind(2),1:nA),b(1:ind(2)),(c+sf)',tol);
        else
                disp('ADDING QUADRATIC SEGMENT.');
                [x,singl_qp_solver_counter,ts]=simp_decomp(D,(c+sf),A(1:ind(2),1:nA),
                b(1:ind(2)),index_matrix);t_s=t_s+ts;
        end
%
%Step 5. Checking For Optimality And Updating Parameters Miou And Eps
%
        t39=clock;
        if   max(abs(ones(size(x'))-x'))<=epsilon
                termination_criterion=1;xx=x;
        else
                x_pr=x;test1=[];
                if sum(test1)==0
                        eps=max(epsilon,nta1*eps);
                else
                        miou=miou*max(V)/(nta2*eps);
                end
        end
        t3t=t3t+t38+etime(clock,t39);disp_matrix=[disp_matrix;main_loop c*x];
end %(WHILE)
t32=clock;t_decIII=etime(t32,t31);tIII_par=t36+max(t_matrix)+t_s+t3t;
```

Appendix G: Problem Generators MATLAB Codes

Linear Problem Generators

Hilbert-Type Problem Generator

```
%This Program Generates The Coefficients For The Hilbert Problems By Given Parameter
N. Please Enter The Values Of Parameter N.

%Step 1. Initialization.
%
n=200;
%
%Step 2. Compute The Coefficients Of Matrix A And B.
%
for i=1:n
        b(i)=0;c(i)=2/(i+1);
        for j=1:n
                b(i)=b(i)+1/(i+j);a(i,j)=1/(i+j);
        end
%
%Step 3. Compute The Coefficients Of Vector C.
%
        for j=2:n
                c(i)=c(i)+1/(i+j);
        end
```

```
end
cc=c;aa=a;
c=[c zeros(1,n)];a=[a -eye(n)];b=b';
%
%Step 4. Compute The Components Of Vector U.
%
u=1.1*ones(2*n,1);
a=sparse(a);b=sparse(b);c=sparse(c);u=sparse(u);
%save
tt1=clock;ttt1=etime(clock,tt1);tt2=clock;ttt2=etime(clock,tt2);
tt3=clock;ttt3=etime(clock,tt3);tt4=clock;ttt4=etime(clock,tt4);
tt5=clock;ttt5=etime(clock,tt5);tt6=clock;ttt6=etime(clock,tt6);
tt7=clock;ttt7=etime(clock,tt7);tt8=clock;ttt8=etime(clock,tt8);
tt9=clock;ttt9=etime(clock,tt9);tt10=clock;ttt10=etime(clock,tt10);
tt11=clock;ttt11=etime(clock,tt11);tt12=clock;ttt12=etime(clock,tt12);
flops(0)
[sol_kar,iter_kar,time_kar,m_kar]=karm93dual([sparse(aa') -speye(n) speye(n)],
sparse(cc'), [sparse(-b') 20*sparse(ones(1,n)) sparse(zeros(1,n))], 20*sparse(ones(3*n,1)),
0.0000005,1e-8,1e-8)
flops_kar_dual=flops
flops(0)
[primalsol,ITER_ALG_A,ITER_ALG_B,TIME1,TIME2,TIME,matrix_dip]=karmar93(a,b,
c,u,0.0000005,0.0000005,1e-8,1e-8)
flops_kar=flops
flops(0)
[sol_barnes,iter_barnes,time_barnes,m_barnes]=barnes(a,b,c,0.0000005)
flops_barns=flops,
flops(0)
[sol_inf,iter_inf,time_inf,m_inf]=infmethod(a,b,c,0.0000005);
```

```
flops_inf=flops,
flops(0)
[sol_cor,iter_cor,time_cor,m_cor]=pdcorpred(a,b,c',0.0000005)
flops_predcorr=flops,
```

Klee-Milty Problem Generator

```
%This Program Generates The Coefficients For The Klee-Minty Problems By Given
Parameters M And N. Please Enter The Values Of Parameters M And N.

%Step 1. Initialization.
%
m=0.4; n=50;b=ones(n,1);c=[1,2*n];d=[1,n];
%
%Step 2. Compute The Components Of Vector C.
%
for j=1:n
        d(j)=-m^(n-j);
end
c=[d zeros(1,n)];
%
%Step 3. Compute The Components Of Matrix A.
%
for i=1:n
        for j=1:n
                if j>i,
                        e(i,j)=0;
                elseif j==i,
                        e(i,j)=1;
```

```
            else,
                e(i,j)=2*m^(i-j);
            end
        end
end
[a1,a2]=percentage(e); a=[e eye(n)];[o,p]=size(a);
%
%Step 5. Compute The Components Of Vector U.
%
u=[0.001*ones(1,n-1) 1 ones(1,n)]';
a=sparse(a);b=sparse(b);c=sparse(c);u=sparse(u);
```

Linear Ordering Problem Generator

```
%This Program Gegerates The Coefficients For The Linear Ordering Problems By Given
%Parameter N. Please Give The Value Of Parameter N.

%Step 1. Initialization.
%
n=10;smax=2*n-1;constr1_counter=0;constr2_counter=0;constr3_counter=0;
mat1=zeros(n,n);mat2=zeros(n,n);mat3=zeros(n,n);
%
%Step 2. Compute The Components Of Matrix A.
%
for s=3:smax
    for i=1:n
        for j=1:n
            if i~=j & i+j==s & j>i,
                mat1(i,j)=1;mat1(j,i)=1;
```

```
                    if any(any(mat1))==1
                        constr1_counter=constr1_counter+1;
                            for ii=1:n
                                con1=[con1 mat1(ii,:)];
                            end
                        a=[a;con1];con1=[];mat1=zeros(n,n);
                    end
                end
            end
        end
end
for i=1:n
    for j=1:n
        for k=1:n
            if 1<=i & i<j & j<k & k<=n
                mat2(i,j)=mat2(i,j)+1;mat2(j,k)=mat2(j,k)+1;
                mat2(k,i)=mat2(k,i)+1;
            end
            if any(any(mat2))==1
                constr2_counter=constr2_counter+1;
                for jj=1:n
                    con2=[con2 mat2(jj,:)];
                end
                a=[a;con2];con2=[];mat2=zeros(n,n);
            end
        end
    end
end
for i=1:n
```

```
        for j=1:n
                for k=1:n
                        if 1<=i & i<j & j<=k & k<=n
                                mat3(j,i)=mat3(j,i)+1;mat3(i,k)=mat3(i,k)+1;
                                mat3(k,j)=mat3(k,j)+1;
                        end
                        if any(any(mat3))==1
                                constr3_counter=constr3_counter+1;
                                for kk=1:n
                                        con3=[con3 mat3(kk,:)];
                                end
                                a=[a;con3];con3=[];mat3=zeros(n,n);
                        end
                end
        end
end
[ma,na]=size(a);
dim1=constr2_counter+constr3_counter;dim2=constr1_counter+dim1;a=[a(:,2:na)
eye(dim2)];[o,p]=size(a);
%
%Step 3. Compute The Components Of Vector B.
%
b=[ones(1,constr1_counter) 2*ones(1,dim1)]';
%
%Step 4. Compute The Components Of Vector C.
%
c=-rand(1,n^2+dim2-1);
%
%Step 5. Compute The Components Of Vector U.
```

```
%
u=2*ones(n^2+dim2-1,1);
a=sparse(a);b=sparse(b);c=sparse(c);u=sparse(u);
```

Random Problem Generator

```
%This Program Generates The Coefficients For The Hilbert Problems By Given Parameter
%N. Please Enter The Values Of Parameter N.

%Step 1. Initialization.
%
n=10;
%
%Step 2. Compute The Coefficients Of Matrix A And B.
%
a=10*rand(2*n/3,n);b=a*ones(n,1);
%
%Step 3. Compute The Coefficients Of Vector C.
%
c=10*rand(1,n);cc=c;aa=a;
%c=[c zeros(1,n)];a=[a -eye(n)];b=b';
%
%Step 4. Compute The Components Of Vector U.
%
u=10*ones(n,1);
a=sparse(a);b=sparse(b);c=sparse(c);u=sparse(u);
```

Quadratic Problem Generator

```
%This Program Generates Separable and Non-Separable Quadratic Problems of Given Size,
Sparsity and Condition Number of the Matrix of the Quadratic Coefficients

m=10;n=20;cond=100;d=20*sprandn(m,n,0.5,1/cond);x=sparse(ones(n,1));
Q=20*sprandsym(n,0.12,1/cond,1);
%Q=speye(n);
p=sparse((20*rand(1,n)-10)');f=d*x;l=sparse(zeros(n,1));u=sparse(20*ones(n,1));
```

Decomposition Problem Generator

```
%This Is A Generator For Block Diagonal Decomposition Problems Proposed By
%Mangasarian (1981).

%Index_matrix Must Be Entered Interactively
%
index_matrix=[5 10;5 10]
%index_matrix=[5 10;5 10;5 10;5 10];
%index_matrix=[5 10;5 10;5 10;5 10;5 10;5 10;5 10;5 10]
mlink=2;[m,n]=size(index_matrix);M=30;
size_dec_problem=sum(index_matrix);nlink=size_dec_problem(2);
%
% Initializing The Decomposition Problem
%
b=[];A=[];c=[];c_temp=0;index_matrix_temp=[];
```

```
%
%Generating Matrix A
%
for i=1:m
        if i==1
                A=[round(100*rand(index_matrix(1,1),index_matrix(1,2)))
zeros(index_matrix(1,1),size_dec_problem(2)-index_matrix(1,2))];
        else
                A=[A;zeros(index_matrix(i,1),sum(index_matrix(1:i-1,2)))
round(100*rand(index_matrix(i,1),index_matrix(i,2)))
zeros(index_matrix(i,1),size_dec_problem(2)-index_matrix(i,2)-sum(index_matrix(1:i-
1,2)))];
        end
end
A=[A;round(100*rand(mlink,nlink))];
%
% Computing Vectors B And C
%
A_sum=sum(A');
for i=1:size_dec_problem(1)+mlink
        if A_sum(i)>0
                b=[b;A_sum(i)];
        else
                b=[b;2*A_sum(i)-1];
        end
end
for i=1:size_dec_problem(2)
        for j=1:size_dec_problem(1)+mlink
                if A_sum(j)>0
```

```
                    c_temp=c_temp+A(j,i);
                end
        end
        c=[c c_temp];c_temp=0;
end
%
%Generating Lower And Upper Bound Vectors
%
l=zeros(1,size_dec_problem(2));u=M*ones(1,size_dec_problem(2));
u=size_dec_problem(1)*ones(1,size_dec_problem(2));
%
%Generating Index_matrix For The Decomposition Algorithm
%
for i=1:m
        if i==1
                index_matrix_temp=[index_matrix_temp;1 index_matrix(i,1) 1
index_matrix(i,2)];
        else
                index_matrix_temp=[index_matrix_temp;index_matrix_temp(i-1,2)+1
index_matrix_temp(i-1,2)+index_matrix(i,1) index_matrix_temp(i-1,4)+1
index_matrix_temp(i-1,4)+index_matrix(i,2)];
        end
end
A=sparse(A);b=sparse(b);c=sparse(c);index_matrix_temp=sparse(index_matrix_temp);
%
% Solving The Decomposition Problem
c1=clock;etime(clock,c1);c2=clock;etime(clock,c2);c3=clock;etime(clock,c3);
c1=clock;etime(clock,c1);c2=clock;etime(clock,c2);c3=clock;etime(clock,c3);
```

```
flops(0);tl=clock;[xi,iter]=pdcorpred(A,b,c',0.0005);time_li=etime(clock,tl),flops_li=flops,
rr=c*xi
flops(0);[t,t_par,x,m_loop,disp_mat]=decomposition3(A,b,c,l,u,index_matrix_temp),flops
_1=flops,max(abs(A*x'-b))
flops(0);[t,t_par,x,m_loop,disp_mat]=decomposII(A,b,c,index_matrix_temp),flops_2=flop
s,max(abs(A*x-b))
flops(0);[x,m_loop,t,t_par,disp_mat]=decomp_li(A,b,c,u,index_matrix_temp),flops_3=flop
s,max(abs(A*x-b))
flops(0);[x,m_loop,t,t_par,disp_mat]=decomp_li_me(A,b,c,u,index_matrix_temp),flops_3
=flops,max(abs(A*x-b))
```